Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN
Vakgroep Computerwetenschappen
Software Languages Lab

A Logic Meta Programming Foundation for Example-Driven Pattern Detection in Object-Oriented Programs

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Coen De Roover

Academiejaar 2008 - 2009

Promotoren: Prof. Dr. Wolfgang De Meuter en Dr. Johan Brichau

ABSTRACT

The growing number of tools for detecting user-specified software patterns is testament to their valuable applications throughout the development process. In these applications, user-specified software patterns describe code that exhibits characteristics of interest. For instance, violations of the protocol an API expects to be adhered to.

Logic formulas can be used as expressive and descriptive pattern specifications. This merely requires reifying the program under investigation such that variables can range over its elements. Executing a proof procedure will establish whether program elements exhibit the characteristics specified in a formula. However, we have observed that such formulas become convoluted and operational in nature when developers attempt to ensure all instances of a pattern are recalled.

As the behavioral characteristics of a pattern can be implemented in different ways, the corresponding formula either has to enumerate each implementation variant or describe the machine-verifiable behavior shared by these variants. The former formulas are relatively descriptive, but only recall the implementation variants that are enumerated. The latter formulas recall all implementations of the specified behavior, but do so by quantifying over information about the program's behavior. This exposes developers to the intricate details of the program analyses that compute this information.

We have reconciled both approaches by embedding source code excerpts in logic formulas. These excerpts exemplify the prototypical implementation of a pattern's characteristics —thus ensuring the resulting specifications are descriptive. They are specified in the concrete syntax of the base program augmented with logic variables. To ensure that all implementation variants are recalled, these excerpts are matched against behavioral program information according to multiple matching strategies that vary in leniency. Each match is quantified by the extent to which it exhibits the specified characteristics. The smaller this extent, the more likely the match is a false positive. This establishes a ranking which facilitates assessing a large amount of matches. A logic of quantified truth provides the theoretical foundation for this ranking.

Unique to our matching process is that it incorporates whole-program analyses in its comparison of individual program elements. A semantic analysis ensures correctness. To compare an unqualified and fully qualified type, for instance, a semantic analysis takes the import declarations into account of the compilation units in which they reside. A points-to analysis increases the amount of implementation variants that are recalled. When expressions are compared, for instance, syntactic deviations are allowed as long as they may evaluate to the same object at run-time.

The resulting example-driven approach to pattern detection recalls the implicit implementation variants (i.e. those that are implied by the semantics of the programming language) of a machine-verifiable characteristic specified as a code excerpt that exemplifies its prototypical implementation.

SAMENVATTING

Het groeiend aantal tools voor het detecteren van door ontwikkelaars gespecificeerde software-patronen getuigt van de waardevolle toepassingen van patroondetectie doorheen het ontwikkelingsproces. In deze toepassingen beschrijft een softwarepatroon telkens code die aan bepaalde eigenschappen voldoet. Een patroon kan bijvoorbeeld schendingen beschrijven van het protocol dat gevolgd moet worden bij het aanroepen van een API.

Logische formules lenen zich tot descriptieve specificaties van zulke patronen. Hiertoe is slechts een reïficatie van het programma vereist zodat formules over de elementen van het programma kunnen kwantificeren. Het uitvoeren van een bewijsprocedure bepaalt dan of elementen van het programma voldoen aan de geformuleerde eigenschappen. We hebben echter vastgesteld dat zulke formules eerder operationeel en verre van descriptief dreigen te worden wanneer ontwikkelaars ervoor trachten te zorgen dat alle instanties van het patroon gedetecteerd worden.

Aangezien de eigenschappen van een patroon op verschillende manieren geïmplementeerd kunnen worden, moet de corresponderende formule ofwel elke implementatievariant opsommen ofwel het gemeenschappelijke gedrag van deze varianten beschrijven. De eerstgenoemde formules zijn relatief descriptief, maar kunnen slechts de opgesomde implementatievarianten detecteren. De laatstgenoemde formules kunnen verschillende implementaties van het gespecificeerde gedrag detecteren, maar moeten hiervoor kwantificeren over informatie die het eigenlijk gedrag van het programma beschrijft. Hierdoor worden ontwikkelaars blootgesteld aan verre van eenvoudige programma-analyses die deze informatie berekenen.

Door broncodemallen te integreren in logische formules hebben we beide opties met elkaar verzoend. Deze van logische variabelen voorziene fragmenten broncode fungeren als een voorbeeld van de prototypische implementatie van de eigenschappen van een patroon. Als voorbeelden van een implementatie waarborgen zij de descriptiviteit van de resulterende patroonspecificaties. Opdat implementatievarianten gedetecteerd zouden worden, vergelijken we zulke broncodemallen met informatie over het gedrag van het programma volgens meerdere strategieën die verschillen in striktheid. Een zogenaamde match bestaat uit bindingen voor de logische variabelen die de broncodemal vervolledigen. Voor elke match kwantificeren we de mate waarin deze aan de gespecificeerde eigenschappen voldoet. Geringe mates zijn indicatoren van valse positieven. De op deze manier verwezenlijkte rangschikking vergemakkelijkt het inspecteren van een groot aantal matchen. Een logica van gekwantificeerde waarheid ondersteunt deze rangschikking theoretisch.

Uniek aan bovenstaand vergelijkingsproces is dat analyses die informatie over het gedrag van het volledige programma berekenen, geconsulteerd worden bij het vergelijken van individuele programma-elementen. De correctheid van matches wordt gevrijwaard door een semantische analyse. Bij het vergelijken van een ongekwalifi-

ceerd type met een volledig gekwalificeerd type, houdt de semantische analyse bijvoorbeeld rekening met de import-declaraties van de compilatie-eenheden waarin beiden zich bevinden. Een points-to analyse verhoogt het aantal implementatievarianten dat gedetecteerd wordt. Bij het vergelijken van expressies zijn syntactische afwijkingen bijvoorbeeld toegestaan, op voorwaarde dat de expressies kunnen evalueren naar eenzelfde object tijdens een uitvoering van het programma.

De resulterende voorbeeldgedreven patroondetectietechniek detecteert impliciete implementatievarianten (deze die volgen uit de semantiek van de programmeertaal) van een patrooneigenschap die gespecificeerd is als een fragment broncode dat fungeert als voorbeeld van de prototypische implementatie van de eigenschap.

ACKNOWLEDGEMENTS

I am greatly indebted to my promotor Prof. Wolfgang De Meuter (Vrije Universiteit Brussel) and co-promotor Dr. Johan Brichau (Université catholique de Louvain). Johan has been my partner in crime from the moment I was ready to dive into his implementation of the SOUL interpreter. I have truly enjoyed our joint work on logic meta programming and I am looking forward to continuing this fruitful collaboration. Despite my stubbornness, Wolfgang never ceased to offer indispensable advise —from the years that preceded my first keystrokes in "`thesis.tex`" to mere hours before I was presenting "`private_defense.keynote`". Over the years, his interventions have been crucial to my evolution from a student to a researcher. It is impossible to thank both Wolfgang and Johan properly for their tremendous support, patience and the countless hours they have spent proofreading each chapter. I have not only come to appreciate them as outstanding advisors, but also as friends.

I sincerely thank the members of my jury for their insightful comments and the significant time they have invested: Prof. Michael W. Godfrey (University of Waterloo), Prof. Ralf Lämmel (Universität Koblenz-Landau), Prof. Viviane Jonckers (Vrije Universiteit Brussel), Prof. Dirk Vermeir (Vrije Universiteit Brussel) and Prof. Theo D'Hondt (Vrije Universiteit Brussel). Next time, I will try to be more concise. During my years as an undergraduate student, I had the pleasure of experiencing the internal jury members as gifted teachers who sparked my interests in software engineering, logic programming and language engineering respectively —the confluence of which my research is situated in.

I wish to extend my sincerest gratitude to Prof. Theo D'Hondt in particular, for welcoming me in the intellectually stimulating environment he has founded and fostered in his own intriguing style. Working at his laboratory has been an unparalleled and extremely gratifying experience. The amounts of trust and freedom to explore that are granted by Theo makes teaching practical sessions for his classes a joy. Moreover, I can only dream of ever mastering all of the clever insights that are delightfully tucked away in the corners of his virtual machines.

Space restrictions prohibit enumerating all of the present and former colleagues in the department with whom I have shared tear jearking laughs and the occasional drink in the KultuurKaffee. A generic, but sincere "*Thanks guys!*" will have to do. However, sharing a drink with our secretaries Lydie Seghers, Simonne De Schrijver and Brigitte Beyens is long overdue. They have helped me out with administrative issues on countless occasions.

My colleagues Charlotte Herzeel and Carlos Noguera deserve a special mention. I had the pleasure of co-authoring a paper on template terms with Carlos before he joined the lab. Now that we have become office mates, I am looking forward to similar endeavors. Charlotte meticulously proofread large chunks of my

dissertation —a huge favor that I hope to return soon. I am equally indebted to Christophe Scholliers, Yves Vandriessche and Frederik Vanden Berghe for sharing and eventually taking over my teaching duties while I was writing.

I would also like to thank those I have collaborated with on logic meta programming in one form or another: Johan Brichau, Roel Wuyts, Johan Fabry, Andy Kellens, Kim Mens, Sofie Goderis, Isabel Michiels, Charlotte Herzeel and Kris Gybels. It is Kris' diligent guidance of my master's thesis that got me interested in research in the first place. My gratitude extends to Tom Van Cutsem, Bruno Defraine, Andy Kellens and Peter Ebraert for not leaving my e-mails which ranged from "*how did you get your dissertation printed?*" to "*which caterer did you order?*" unanswered.

Thanks also go to everyone who helped me move to a new apartment while I was writing: Elisa Gonzalez Boix, Christophe Scholliers, Wolfgang De Meuter, Sven Casteleyn, Pascal Costanza, Alfredo Cádiz and Kris Gybels. Never before had I seen a truckload of Swedish furniture being carried up three floors and being assembled on the same evening! My apologies for the thumbs that suffered in the process.

Words alone will not suffice to express my gratitude towards my parents and family. My mom is a saint —it is odd that my little sisters Jolien and Liesbeth are such endearing devils.

Finally, my sincerest thanks go to Elisa Gonzalez Boix and Rolando Romero Mendíburi. It is their support and listening ear that got me through the difficult, questioning moments.

Coen De Roover
August 2009

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

He said: *"We've figured out a way to make the document creation program look like the punched card reader to the computer. We need to make a copy of your source code deck and store it just like the report you are working on, and then you can use the Selectric typewriter to make your program changes. We need you because you already know how to use the typewriter. I'll help you get everything organized the way you will need it."* The next several days were hectic. All the programmers were learning how to use the document creation program, and there was always a line of people waiting to use the Selectric typewriter down the hall. A few of the programmers were reluctant to make the change. Giving up their punched cards was a big leap; being able to hold their source deck in our hands was important. Also, the ability to hand a coding sheet to Maria in the keypunch room had always been heavily used by those without typing skills. But soon even the non-typists were converted, however reluctantly, to the typewriter. The increased productivity, even for a hunt-and-peck typist, was just too great to ignore.

Programming with Punched Cards
DALE FISK [Fis05]

1.1 Research Context

The era in which a text editor, operated through a typewriter computer terminal, could cause such a stir among programmers has long since passed. Source code is still written in text editors (e.g. EMACS and VIM), but the editors have become aware of the language in which the text is written. The well-defined syntax of a programming language allows them to inspect the source code and assist the programmer in the creative process. Source code is formatted automatically, highlighting keywords and syntax errors. Outlines are presented that ease navigation to a specific section in the code. Completions for expressions are suggested as they are entered. Even the find and replace functionality has become syntax-aware.

Tool Support for the Development Process

At the same time, the development process itself has become more systematic. Tools assist in version management of source code (e.g. SVN and DARCS) and check whether the coding conventions for a programming language are adhered to (e.g. CHECKSTYLE for Java) . Tools are also able to instantiate design patterns [GHJV94] to an implementation (e.g. ARGOUML) or recognize bug patterns [All02] in existing implementations (e.g. PMD and FINDBUGS for Java). The former document proven solutions to recurring design problems, while the latter document implementations that often lead to bugs. All of these tools offer support for the development process.

Integration of Tools within Development Environments

Editors have grown into integrated development environments (IDEs, e.g. VISUAL STUDIO), integrating functionality from various source code manipulating tools. Developers can edit, compile and debug their programs in a single environment with a uniform interface. Time freed by not having to switch continuously between separate tools can be spend on the creative process.

More importantly, the environment relates the heterogeneous information offered by the tools it integrates. Implicit links, which would only become apparent after intensive manual comparisons, are explicated. For instance, the origin of a compilation error is highlighted in the source code where it can be corrected. By integrating semantic information about the program, the environment is able to make intelligent suggestions to the developer. The same information allows the code to be navigated swiftly through semantic perspectives. For instance, a declaration can be found for each use of a variable and the overrides of a method can be browsed.

By adopting extensible architectures, integrated development environments have embraced the increase in tool support. Their functionality can be extended by loading a plug-in that integrates the tool within the environment. As a result, all tools can be operated through a relatively uniform interface within a single environment.

In the internet age, developers can browse and install plug-ins from on-line plug-in repositories. In fact, a recent survey by Forrester research on IDE usage [HSD08] found that 39% of 703 surveyed developers use three to five plug-ins regularly. Of the developers using the ECLIPSE IDE for Java, 81% even cited using three or more plug-ins regularly. Factors holding back plug-in adoption were surveyed as well. For 61% of the correspondents, lack of documentation and training was cited as the biggest factor. At 28%, poor integration was cited 6th as the biggest discouraging factor. Although the tools that support the development process are diverse, their integration within the host environment encourages their use.

Application-Specific Support for the Development Process

Application-specific support for the development process is of special interest. It is, for instance, needed by a development team that has agreed upon a common programming style and desires tool support to enforce it. It is also needed by a developer who, upon discovering suboptimal code, wants to verify automatically whether or not its effects are more widespread throughout an application. It is

needed as well by developers who want to detect violations against the protocol an in-house application programmer interface (API) expects to be adhered to. It is furthermore indispensable whenever code has to be navigated through application-specific perspectives. For instance, to inspect the methods that cross architectural application layer boundaries (e.g. by invoking the persistency layer from the presentation layer) or break the desired abstraction into layers (e.g. by querying the database directly rather than going through the persistency layer). The boundaries of these layers are inherently application specific.

Application-Specific Support through Pattern Detection

The need to identify program elements that exhibit particular characteristics is common to the development support desired above. Following the liberal definition of a *pattern* by Riehle et al. [RZ96], we will refer to a tool that offers this functionality as a *pattern detection tool*[1].

The kind of characteristics supported by a pattern detection tool varies with the use it is intended for. Characteristics concerning the execution order of instructions, for instance, are essential to tools intended for checking protocol conformance. Characteristics concerning the relations among object-oriented entities (e.g. ancestorship and overriding) are, in contrast, essential to tools intended for program navigation.

To support application-specific uses, however, pattern detection tools cannot be limited to a fixed catalogue of predefined patterns.

Many tools have therefore adopted an open architecture. The popular bug pattern detection tools PMD, FINDBUGS and CHECKSTYLE support custom traversals of a program's code to search for user-specified patterns. However, this approach to extensibility comes at a high price. Users of the aforementioned tools are required to extend a low-level implementation of the Visitor design pattern that traverses the program's code.[2] This means users end up implementing the search for a bug rather than specifying the bug they are looking for. Moreover, they are exposed to the tool's internal representation of the program under investigation.

Such problems are, in contrast, not associated with pattern detection tools that accept a descriptive specification of the characteristics of interest and report all program elements that match this specification. In addition to a program representation, these tools feature a specification language and a detection mechanism. Each configuration in this design space is tailored to the intended use of the tool.

To support characteristics concerning the execution order of instructions, for instance, a pattern detection tool needs a specification language in which key instructions and their sequencing can be described. Its detection mechanism has to determine whether a specified instruction sequence may be observed during an execution of the program. To this end, it has to analyze a representation of the possible executions of the program under investigation (e.g. a control flow graph).

Apart from the more conventional ones enumerated in the previous section, some very original applications of user-specified pattern detection have been identified following the shift towards declarative specification languages. Examples include co-evolving the design and documentation of object-oriented programs [Wuy01], improving the coverage of program analyses by resolving idiomatic

[1] Precise definitions of software patterns and their machine-verifiable characteristics follow in Chapter 2.

[2] PMD also supports XPath queries over an XML representation of the program's code.

uses of reflection [LWL05b] and semantic patching of applications [PLM06]. With every new application that is identified, the use of pattern detection tools will become more widespread among developers.

1.2 Problem Statement

Program analyses [NNH05] that compute precise information about run-time values are, in particular, costly for object-oriented programs due to polymorphism and late binding. Recent advances in performance and scalability have enabled tools to support behavioral pattern characteristics without incurring a computational cost that hinders their integration with an interactive development environment.[3] This opens up new application areas to pattern detection tools. [4]

In spite of their applications, developer adoption of tools that detect user-specified behavioral characteristics is not widespread. Specifying and subsequently assessing the results returned for behavioral characteristics is hard.

Specifying Behavioral Characteristics is Hard

Specifying behavioral characteristics poses problems. Such characteristics not only concern the execution order of instructions, but also the flow of run-time values operated on by instructions. The problems are related to the numerous ways in which a behavioral characteristic can be implemented. For example, an instance field `author` in Java can be referenced through the expression `author` or through the expression `this.author`. In addition, either variable `person` or `artist` may be used to refer to its value after the assignments `person = this.author` and `artist = getAuthor()` respectively[5].

If detecting implementation variants of a behavioral characteristic is not supported, users have to resort to enumerating each variant in a specification. This results in convoluted specifications. Moreover, enumerating all possible variants is impossible in practice. Many program elements that exhibit the desired characteristics will not be recognized. Finally, program elements might even be identified erroneously if the user fails to consider the semantics of the programming language.[6]

Existing pattern detection tools focus on the implementation variants of one kind of behavioral characteristics. Just one of the surveyed tools that support execution order characteristics (i.e. DEEPWEAVER [FKI+07]), for instance, offers support for the implementation variants of the characteristics concerning run-time values. However, its users must relate the results from different program analyses manually. Each analysis supports a different behavioral characteristic. As a result, its users are exposed to the intricate details of each analysis.

If behavioral characteristics cannot be specified at a higher level of abstraction than the program analyses that enable the detection of their implementation vari-

[3] For instance, by adopting clever representations of the computed information [LWL+05a, Lho06] that save both time and memory. Or by using demand-driven [SR05] (i.e. only computing the results a tool needs), incremental [SR05] (i.e. computing changes in results given a change in input) and refinement-based [Sri07] (i.e. refining results from a less costly analysis within a time budget) algorithms.

[4] For instance, a very precise points-to analysis [WL04] has been instrumental in detecting tainted data problems [MLL05] (i.e. the unchecked passing of user data to sensitive methods such as the infamous SQL injection vulnerabilities).

[5] In case method `getAuthor()` is a getter method for the field author.

[6] Scoping rules, for instance. Within the method `m(Object author)`, the expression `author` refers to the parameter of the method rather than the field desired in the enumeration above.

ants, users are exposed to the program analysis domain. This adds to the already steep learning curve associated with a pattern detection tool. Distinguishing true from false references to the `author` field, for instance, requires the results of a semantic analysis [ASU86]. Such an analysis relates identifiers to their definition according to scoping rules. An alias analysis [Hin01] is, on the other hand, required to determine that the expression `person` evaluates to the value of the field. All of these analyses stem from the optimizing compiler domain.

Assessing Results for Behavioral Characteristics is Hard

Users have to be aware of the extent to which a tool is able to assert the presence of a behavioral characteristic. Not only does this extent vary among different tools, but it may also vary among the program elements identified by a single tool. These differences originate from differences in the precision of the analyses that are employed by each tool.

Consider once more the various implementations referring to the value of the `author` field. Implementation `this.author` requires a semantic analysis, while implementations `person` and `artist` require an alias analysis. Without an alias analysis, the latter will not be detected at all. Moreover, depending on whether a may-alias or must-alias analysis is used, the aliasing of these variables is asserted to different extents. A may-alias analysis determines whether they may alias during an execution of the program. A must-alias analysis determines whether they must alias during every execution —which is more expensive. Some tools might therefore reserve the must-alias analysis for local variables and rely on the less expensive may-alias analysis for others. As a result, the extents to which reported program elements exhibit a specified behavioral characteristic may vary greatly. *Assessing the extent to which a reported program element exhibits a specified behavioral characteristic requires detailed knowledge about a tool's enabling analyses.*

On a pragmatic note, the results of a tool often include elements from intermediate representations employed by its enabling analyses. This is the case for 8 out of the 10 tools that support behavioral characteristics surveyed in Chapter 3. Examples include elements from three-address representations in which all instructions take the form of two operands, an operation and a result.[7] As a result, users are often exposed to the implementation details of a tool's enabling analyses as well. Integration with development environments is comprised, forcing users to map elements from intermediate representations to the program's code.

No Unified Support for Behavioral and Non-Behavioral Characteristics

Most existing pattern detection tools have been designed for a single, specific use. The specification language, detection mechanism and program representation of each tool are specialized in the characteristics that are essential to its intended use. Other characteristics are often not supported at all (see Table 3.3).[8]

[7] Figure 2.4 depicts a common variant for Java programs. Such intermediate representations simplify the program analyses that enable detecting behavioral characteristics.

[8] None of the tools intended for program navigation [JD03, HVd06], for instance, supports characteristics concerning the execution order of instructions which are important for detecting bugs. Likewise, none of the bug pattern detection tools [ECCH00, BE03, Vol06a] supports characteristics concerning the relations among program entities that are essential for program navigation.

As a result, developers are faced with *specification languages that are as diverse as the intended uses of the pattern detection tools* that offer them (see Table 3.1 for an overview). Consider the tools tailored to characteristics concerning the execution order of instructions. Their specification languages already include finite state machines [ECCH00, BE03], regular expressions [DdMS02, LRY+04, VEdM06] and temporal logic formulae [LdM01] over control flow graphs. Tools tailored to characteristics concerning the syntax of instructions feature completely different specification languages. Examples include source code fragments [Pau92, SV98, Mos05], various logic programming languages [BCD+89, Cre97, RKA06, AK07] and regular expressions over abstract syntax trees [GAM96].

Only a tendency towards logic formalisms (i.e. about 13 out of the 26 declarative specification languages surveyed in Chapter 3 feature logic propositions in the classical syntax or a custom surface syntax) somewhat lessens the problem which the diversity among specification languages poses to users.

Currently, users have only two options to detect a heterogeneously characterized pattern. The first is to divide its specification over several tools specialized in a particular characteristic. Users have to relate their results manually. The alternative is to use a general-purpose tool with a specification language that is an amalgamation of languages individually designed to express one kind of characteristic. *A unified specification language for a general-purpose tool has not been investigated yet.*

1.3 A Logic Meta Programming Foundation for Example-Driven Pattern Detection

We have investigated which configurations of a specification language, detection mechanism and program representation result in a *general-purpose* pattern detection tool that supports the detection of *behavioral as well as non-behavioral characteristics* using *descriptive and declarative* specifications in a *unified* language.

The pattern detection tool that corresponds to these configurations is a *general-purpose* tool that can be applied throughout the development process. Section 2.6 formulates the criteria for this tool to fulfill. They are summarized in Table 2.1. Each criterion is motivated in response to the problems identified above. To ease the problems associated with implementation variants, for instance, we require the *specification language* to support expressing *explicit* points of variation among the variants and require the *detection mechanism* to support detecting *implicit* points of variation among the variants. The latter represent different implementations of the same characteristic, while the former represent variations among the characteristics that should be detected.

In this dissertation, we present a logic meta programming foundation for detecting software patterns in an example-driven way. The resulting example-driven approach to pattern detection fulfills the aforementioned criteria for a general-purpose pattern detection tool. Its contributions to the state of the art will be detailed in our concluding chapter.

In this section, we aim to give a brief overview of the approach. A detailed overview is given in Chapter 4 which includes some characteristic examples. To make the idea of example-driven pattern detection somewhat more tangible, we invite the reader to briefly examine Figure 4.7. It depicts a specification for the getter method best practice pattern. Figure 4.12 depicts the corresponding methods

from Figure 4.8 that are identified as getter methods by our research prototype.

Our example-driven approach has its roots in the *logic meta programming* [Wuy01] approach to pattern detection. It is widespread in the literature (e.g. [BCD+89, KP96, Cre97, RD99, Wuy01, JD03, RKA06, AK07, FKI+07] and [MLL05, HVd06, CGM06b] use variants of Prolog [EK76] and Datalog [CGT89] respectively). In this approach, a machine-executable proof procedure for a logic is adopted as detection mechanism —giving rise to a programming language. A pattern specification consists of logic conditions that range over a reified program representation. Therefore, we can speak of a logic meta program.

The following overview is structured according to the previously identified dimensions in the design of pattern detection tools. We limit ourselves to the differences with respect to the logic meta programming approach.

1.3.1 Specification Language

To support *example-based specifications*, the specification language of our approach embeds source code excerpts in logic queries. The source code excerpts correspond to the *prototypical implementation* of the essential characteristics of a pattern. They are specified in the *concrete syntax* of the program under investigation, extended with logic variables. Logic variables within a source code excerpt demarcate points of variation. The resulting specifications are highly descriptive, while their syntax is already familiar to developers.

Source code excerpts can be combined with other excerpts and logic conditions using logic connectives. Multiple occurrences of a variable are supported, within excerpts as well as conditions (e.g. variable *?expression* on lines 3 and 4 of Figure 4.6). This facilitate expressing the explicit variation points of a pattern (i.e. variations in its characteristics).

1.3.2 Detection Mechanism

Following the logic meta programming approach, the detection mechanism of our approach is based on the machine-executable proof procedure for Prolog [EK76]: resolution [Rob65]. The differences are described below.

It Realizes the Example-Based Semantics of Source Code Excerpts

The detection mechanism of our approach realizes the *example-based semantics* of the source code excerpts in a query. The same excerpt can exemplify both non-behavioral as well as behavioral characteristics of the prototypical implementation of a pattern. The detection mechanism has to account for all possibilities. Several example-based interpretations are therefore considered for each source code excerpt.

In essence, each example-based interpretation transforms a source code excerpt into a logic query. To support behavioral characteristics concerning the execution order of instructions, for instance, one interpretation transforms excerpts into queries over control flow graphs. Each transformation is specified as a logic rule. This allows users to define additional example-based interpretations.

It Quantifies Results through Fuzzy Logic

The results reported by the detection mechanism of our approach consist of a program element and a *quantification of the extent to which it exhibits the specified characteristics*. The reported quantifications are intended to facilitate assessing a large amount of results.[9] To this end, it employs a resolution procedure for a *fuzzy logic*. This is a logic of quantified truth based on fuzzy set theory [Zad65]. The truth of logic propositions, between absolutely falsity and absolute truth, is quantified.

In fuzzy logic programming, a truth degree is associated with each logic rule. They express the confidence in its conclusion given the absolute truth of the conditions in its body. We use this feature to establish a *ranking among the example-based interpretations* of source code excerpts. The right column in the upper left window in Figure 4.12, for instance, lists the truth degrees for each solution to the specification in the bottom left window —solutions stem from Figure 4.8.

The actual quantification of a logic goal (e.g. a condition in a query) varies with each fuzzy Prolog. The particular variant used in our approach is similar to the one of F-PROLOG [LL90], with the exception that it also quantifies unification (see next section). This allows us to establish a *ranking among the results reported by a single example-based interpretation* of a source code excerpt.

It Employs a Domain-Specific Unification Procedure

The detection mechanism of our approach employs a *domain-specific unification* procedure. It differs from the general-purpose unification procedure on the following points.

- To support *implicit variation points* of a pattern (i.e. different implementations of the same characteristic), *whole-program analyses* are incorporated in the comparison of *individual program elements*. Users benefit from their results without being exposed to their intricate details —even when defining additional example-based interpretations for source code excerpts.[10]

  A *semantic analysis* ensures correctness. For qualified and unqualified names of types, import declarations are taken into account. A *points-to analysis* enhances identification efficacy. When expressions are compared, syntactic deviations are allowed as long as they may-alias at run-time. Using the points-to analysis to resolve late binding, method invocations unify with their possible target method declarations.

- Unification results either in failure, or in a unification degree. Unification degrees are propagated by the fuzzy resolution procedure to the reported results.

  Through these degrees, our approach communicates information about the program analyses used in a domain-specific unification. For instance, whether unifying variables required a semantic analysis or a points-to analysis. The former is the case for variables that are bound to an unqualified name and the fully qualified name of the same type respectively. The latter is

[9] This is the sole intent of the reported quantifications. In particular, they should not be confused with probabilities.

[10] Advanced users can define additional domain-specific extensions of the general-purpose unification procedure through a tool. It is only here that they are exposed to the details of the program analyses.

the case for variables that are bound to two expressions that may alias in an execution of the program.

- To enable the natural use of unification to quantify over the program representation, a *reified program element* (e.g. the abstract syntax tree node for the qualified name of a type `x.y`) *unifies with a structurally equivalent compound term* (e.g. `qualifiedName(simpleName('x'),simpleName('y'))`) —even if the reified version of the program element is not a compound term. A reification is needed by the logic meta programming approach to support querying the program representation using logic conditions.

  Our approach forgoes the prevalent transcription to compound terms (e.g. a database of logic facts) and queries the program representation directly (see next section). Our unification procedure avoids that queries that have to quantify over the program representation become convoluted. Otherwise, compounds such as `qualifiedName(?qualifier,simpleName('y'))` could not be used to identify types of which `y` is the last part of the qualified name. Our approach lends the reified name an alternative, compound-based representation on-the-fly.

1.3.3 Program Representation

The program representation of our approach *includes both behavioral and non-behavioral information* about the program under investigation. However, our logic specifications range only over abstract syntax tree nodes. In other words, *only abstract syntax trees are reified* in our approach. This way, the user is shielded from the intricate details of program analyses. It also precludes elements from intermediate program representations from popping up in the reported results. This, in turn, facilitates user assessment and integration with other tools.

As mentioned in the previous section, our approach does not reify abstract syntax tree nodes as compound terms. Rather, the nodes are kept as is (e.g. objects). This renders reconstituting the actual AST nodes from their reified counterparts trivial at any point in the proof procedure. In particular, incorporating whole-program analyses in the unification of individual abstract syntax tree nodes is facilitated. The context within the program of each node can be obtained easily (e.g. parent node or parent method declaration).

Our approach is agnostic with respect to the concrete logic meta programming language that is used as its foundation. In our concrete instantiation, we have opted for a language in which objects are first-class values: the Smalltalk-Prolog hybrid SOUL [Wuy01].[11] Such a hybrid opens the door to the imperative paradigm in specifications wherever it is more convenient.

1.4 Dissertation Outline

Chapter 2: Detection of User-Specified Software Patterns This chapter primarily serves to formulate and motivate the criteria for a general-purpose pattern detection tool. However, some background information is provided first.

[11] In fact, within Smalltalk expressions within logic rules, our prototype communicates with a Java Virtual Machine to reify an ECLIPSE workspace. It relies on the interoperability between Java and Smalltalk provided by the JAVACONNECT [Jav] library.

We present detailed definitions for software patterns and their machine-verifiable characteristics. We illustrate that software patterns are more varied than the canonical design patterns lead to believe and enumerate interesting applications of their detection in software engineering. For each machine-verifiable characteristic, we investigate which configurations in the design of a tool support its detection.

Chapter 3: State of the Art in Pattern Detection In this chapter, we survey the state of the art in tools that support the detection of user-specified patterns. The survey is structured according to the characteristics each tool is primarily intended for. The fact that such a structure is possible, already highlights the current lack of general-purpose pattern detection tools.

Chapter 4: An Example-Driven Approach to Pattern Detection This chapter defines our example-driven approach to pattern detection in terms of four cornerstones and their inter-dependencies: logic meta programming, example-based specifications, domain-specific unification, fuzzy logic and open implementations. Each cornerstone is introduced and carefully motivated with respect to the criteria for a general-purpose pattern detection tool it helps fulfill. Their individual contributions are also illustrated on a running example. As a suggestion for their concrete instantiation, we briefly outline the implementation of each cornerstone in our research prototype. The remaining chapters revisit the implementations in detail, except for the last cornerstone which crosscuts all others.

Chapter 5: Instantiating the Logic Meta Programming Cornerstone This chapter discusses the LMP cornerstone as instantiated in the prototype that we will use to validate our approach. Concretely, this LMP instance consists of the Smalltalk-Prolog hybrid SOUL and the CAVA library of predicates for reasoning about Java programs. To illustrate the support offered by LMP for each kind of pattern characteristic, we specify representative patterns as logic queries. Subsequent chapters revisit the same patterns.

Chapter 6: Instantiating the Fuzzy Logic and Domain-Specific Unification Cornerstones
In this chapter, we discuss the instantiations of the fuzzy logic and domain-specific unification cornerstones. Both instantiations adapt the logic meta programming instance discussed in the previous chapter. We specify the representative patterns of the previous chapter as fuzzy logic queries that use domain-specific extensions to the general-purpose unification procedure. This illustrates how both cornerstones improve upon the support for each pattern characteristic offered by the founding cornerstone.

Chapter 7: Instantiating the Example-Based Specification Cornerstone This chapter discusses the example-based specification cornerstone as instantiated in our prototype. We revisit the representative patterns of the previous chapters to demonstrate how this cornerstone enables exemplifying their characteristics by means of a code excerpt that corresponds to their prototypical implementation.

Chapter 8: Validation - Detecting Patterns using Example-Based Queries In this chapter, we validate our example-driven approach to pattern detection by

applying its instantiation to several interesting software patterns. An evaluation on the criteria for a general-purpose pattern detection tool concludes this chapter.

Chapter 9: Conclusion and Future Research Directions This chapter concludes our dissertation. We revisit the problem statement and restate the contributions of our dissertation. Finally, we present interesting directions for future research.

1.5 Supporting Publications

Of the (co-)authored publications that are related to logic meta programming [DGD05, DMG+06, DBD06, MDB+06, HGC+07, DBN+07, BDM07, KBD08, HGC+09, BD09], the following introduce the key ideas of our example-driven approach to pattern detection:

- **Combining Fuzzy Logic and Behavioral Similarity for Non-strict Program Validation** [DBD06]
  Coen De Roover, Johan Brichau and Theo D'Hondt
  Proceedings of the 8th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2006)
  This paper explores extending the unification procedure of an LMP language such that incompatible logic terms unify if they reify Java expressions that are in a may-alias relation. It also proposes using fuzzy logic to discern solutions that required such an extension from the ones that were found under the regular unification procedure. Both are predecessors of the instantiations of the domain-specific unification and fuzzy logic cornerstones discussed in Chapter 6. The more refined unification procedure, for instance, incorporates additional program analyses such as a must-alias and a semantic analysis.

- **Behavioral Similarity Matching using Concrete Source Code Templates in Logic Queries** [DBN+07]
  Coen De Roover, Johan Brichau, Carlos Noguera, Theo D'Hondt and Laurence Duchien
  Proceedings of the 2007 ACM-SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2007)
  This paper explores the use of source code excerpts to exemplify the prototypical implementation of various software patterns. In contrast to the more refined instantiation of the example-based specification cornerstone discussed in Chapter 7, only a single example-based interpretation of the excerpt is considered to resolve the term (i.e. an inter-procedural version of the lexical interpretation discussed in Section 7.2.2).

- **Open Unification for Program Query Languages** [BDM07]
  Johan Brichau, Coen De Roover and Kim Mens
  Proceedings of the 26th International Conference of the Chilean Computer Science Society (SCCC 2007)
  The research artifacts supporting the above publications rely upon the logic predicates of the IRISH library [FM04] to reason about Java programs. The artifact supporting this dissertation relies upon the predicates of the CAVA library instead (cf. Chapter 5). The paper introduces a predecessor of this library for reasoning about

the projects in an Eclipse workspace. In addition, the paper explores two domain-specific unification extensions: one that consults a semantic analysis to unify two reified Java AST nodes (cf. Section 6.4.1) and one that unifies a reified Java AST node with a structurally equivalent compound term (cf. Section 6.4.2). The paper also explores a domain-specific unification procedure for Smalltalk programs. However, in contrast to the unification procedure for Java programs, it does not incorporate any whole-program analyses.

Finally, in [DGD05], we motivated using advanced program analyses to support pattern detection applications of LMP. In [BD09], we discuss the JAVACONNECT library in more detail. It enables Smalltalk applications to invoke methods on any Java object in a running JVM instance. This is how the aforementioned CAVA library is able to reason about the projects in an Eclipse workspace (cf. Chapter 5).

CHAPTER 2

DETECTION OF USER-SPECIFIED SOFTWARE PATTERNS

We initiate our discourse on pattern detection with an overview of its applications in software engineering. We illustrate that for many applications, domain-specific and application-specific patterns preclude the use of a predefined pattern catalogue. Instances of the targeted patterns exhibit non-behavioral, but also behavioral characteristics. While the former concern the program's syntactic and architectural organisation, the latter concern the execution order of its constituents and the values they evaluate to. We identify the key dimensions in the design of a tool that supports the detection of user-specified patterns. We investigate which combinations of a specification language, detection mechanism and representation of the program under investigation support each individual kind of pattern characteristic. We conclude the chapter with an overview of the criteria a general-purpose pattern detection tool should exhibit. These criteria are organized according to the previously identified design dimensions and will be used to assess the state of the art in the next chapter.

2.1 Software Patterns

Following the liberal definition introduced by Riehle et al. [RZ96], we use the term *"pattern"* to denote *"an abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts"*. The form of a pattern consists of *"a number of visible and distinguishable components and their relationships"*. The abstract form of a pattern materializes as a mental concept, according to this definition, from reflection on experience gained from recognizing and comparing its recurring concrete forms. These are referred to as pattern instances. The abstract form can also be used as a template to create new pattern instances in a new context of use. This context is said to constrain the abstract form of the pattern to the concrete form of its instance. It is moreover non-arbitrary in the sense that all contexts in which concrete forms recur share specific commonalities such that it is beneficial to doc-

ument and abstract the concrete forms into a pattern.

"Software patterns" denote patterns of which the form consists of software entities and their relationships. We restrict the scope of this dissertation to software patterns of which the *recognition can be automated*: their abstract form consists of *"machine-identifiable software entities"* and the relationships among these software entities are machine-verifiable. In keeping with the automated pattern recognition setting, we refer to the source of the software entities in the concrete pattern forms as the *"program under investigation"*. Machine-identifiable software entities encompass both higher-level and lower-level constituents of the program under investigation. These range from packages and compilation units over classes, individual statements and expressions to the keywords and identifiers contained therein. The *"machine-verifiable relationships"* can express both *"non-behavioral as well as behavioral characteristics"* of the entities involved in a pattern's concrete form. The former express syntactic and organisational characteristics of the program's source code and architecture respectively. The latter express the relative order in which entities are executed or properties of the values they evaluate to at run-time. The specific characteristics considered in this dissertation are denominated later on.

Well-known software patterns of which the recognition can be automated include, but are not limited to, the following:

- application-specific patterns that capture the programming style and conventions agreed upon by a team of developers (e.g. the coding conventions for Java programs advocated by Sun Microsystems [Mic99] or those checked by the CHECKSTYLE project [Che08]).

- language-specific idioms such as the idiomatic way to enumerate all elements in a collection. Other examples include the idiomatic implementation of double dispatching in single dispatch languages and the idiomatic use of the reflection (see Livshits et al. [LWL05b] for case studies of the latter in Java, while Peel [Pee87] and Coplien [Cop91] represent early collections of advanced idioms for APL and C++ respectively).

- the coarse-grained, but implementation-level object-oriented μ-patterns coined by Gil et al. [GM05, CGM06a] which describe software entities in straightforward structural relations. Examples include the *Implementor*, *Overrider* and *Extender* μ-patterns which describe a class that exclusively implements abstract methods, overrides existing methods and enriches an inherited interface respectively. The *Record* μ-pattern is another example which describes a class without methods and only public fields.

- best practice patterns [Bec96] which capture software engineering best practices such as the use of getter and setter methods to eliminate direct field accesses hindering the evolution of classes. Conversely, bad practice patterns document bad practices. An example is dropping a raised exception by catching the exception without handling it.

- patterns describing bad smells [FBB+99] which are indications of a suboptimal implementation suggesting the need for a thorough refactoring. An example of a bad smell is a method that is too long or has too many parameters.

- language-specific and application-specific bug patterns that could lead to erroneous behavior at run-time. Language-specific bug patterns are not limited to the classical dereferencing of a null pointer or the reading of a closed file, but also include common pitfalls such as the unintended use of an assignment operator in the condition of an `if`-statement. Implementing a custom `hashCode()` method without implementing a custom `equals(Object)` method is another common Java pitfall that could lead to violations of the invariant that equal objects must have equal hash codes (see the *FindBugs* project [HP04], the PMD [PMD08] project and Allen [All02] for comprehensive collections of Java bug patterns). Many application-specific bug patterns amount to violations of the protocol expected by the application programmer interface of a library (e.g. a driver's failure to restore the interrupts it disabled through the API of an operating system kernel as examined by Chou et al [CYC+01]).

- certain security vulnerabilities such as the passing of untrusted data to sensitive methods without appropriate security measures (e.g. the SQL injections common in web-based systems that pass a string as it was received from the user to their database server)

- the seminal design patterns introduced by Gamma et al. [GHJV94] which describe proven object-oriented solutions to common design problems

- pitfalls in the implementation of design patterns [GHJV94] and software architectures. An example of the former are *lapsed listeners*. These are listeners from the *Observer* design pattern that are no longer needed, but are never garbage collected because they never unregistered with their subject (see the Checklipse [Liv05] project for pitfalls related to the *Observer* and *Template Method* design patterns as observed in Eclipse [Liv05]). An example of the latter arises in straightforward implementations of a distributed *Blackboard* architecture [BHS07]. With a permissive control component that always returns the most freshly published data, inconsistencies between knowledge source components potentially arise when these components directly or indirectly alter the state of local data after having it published on the blackboard.

2.2 Machine-Verifiable Pattern Characteristics

In this dissertation, the only relationships among software entities that are considered for the form of a software pattern express its *"machine-verifiable syntactic, structural, control flow or data flow characteristics"*. While the first two are of a non-behavioral nature, the last two are behavioral as they concern the run-time behavior of the program under investigation.

Consider the bug pattern describing the reading from a closed file as an example. Through *syntactic characteristics*, the software entities in its form are restricted to expressions `x.close()` and `y.read()`. Through *data flow characteristics*, the software entities are further restricted to those in which variables `x` and `y` denote the same file. The *control flow characteristics* on the other hand express that the reading from this file follows its closing at run-time.

2.2.1 Syntactic Characteristics

The syntactic characteristics in a pattern's form concern the syntactic structure of source code according to the formal grammar of the programming language it is written in. They restrict the software entities in the form to source code elements belonging to a particular grammatical category such as the one of statements or the one of all productions that are not expressions. Syntactic characteristics also concern the relations, according to the grammar, between source code constructs in the pattern's form. For instance, the aforementioned unintended assignment pattern consists of an `if`-statement and an assignment expression such that the latter is produced by the expression-part of the former's grammar rule. Many low-level software idioms are primarily characterized by syntactic characteristics.

Behavioral characteristics are not easily expressed in terms of syntactic characteristics as the semantics of the programming language must be taken into account. While two variables can be related syntactically through the strings they are identified by, they do not necessarily refer to the same run-time entity. Likewise, the semantics of exception handling and control statements must be accounted for in order to express through syntactic characteristics that two statements may be executed sequentially at run-time.

2.2.2 Structural Characteristics

The structural characteristics in a pattern's form concern the structural organization of the program under investigation. They restrict the entities in the form to *select* software entities that are too *coarse grained* to reconstruct the complete source code of the program under investigation, but nonetheless offer a *birds-eye-view of its organisation*. Examples include packages, compilation units and the types defined therein. The machine-verifiable relations among these entities range from a generic containment relation over inheritance relations between types to invocation relations between methods. Many of the aforementioned μ-patterns [GM05, CGM06a] and especially the structural rather than the creational or behavioral design patterns [GHJV94] are primarily characterized by structural characteristics.

Structural characteristics can be expressed in terms of syntactic characteristics. While this is straightforward compared to the expression of behavioral characteristics, it results in large expressions with idiomatic recurring parts because structural entities and relations must be derived from the complete syntactic structure of the program's source code.

2.2.3 Control Flow Characteristics

The control flow characteristics in a pattern's form concern the order in which selected software entities are executed at run-time. They stipulate how the execution of key instructions, such as the reading and closing of a file, must be sequenced at run-time. The software entities in a particular sequencing relation are often syntactically dispersed throughout the program's source code. Examples of straightforward machine-verifiable sequencing relations are consecutive execution and transitive reachability. Other relations impose restrictions on the kind of instructions that can occur in between two instructions. The most refined relations express that sequences of executed instructions fit a certain form. As the behavior of a program

possibly varies each time it is executed, these forms can be required to hold for one, some or all run-time executions of the program. Dwyer et al. [DAC99] present a taxonomy of sequencing relations and compare common formalisms to express them.

Control flow characteristics are common in the form of bug patterns. As illustrated above, they are usually complemented by syntactic and data flow characteristics. These serve the identification of the key instructions in a program's execution upon which sequencing constraints are imposed. Without data flow characteristics, sequencing constraints are limited in their ability to relate instructions. False positives might be reported. This is the case if the receiver variable of the `x.close()` invocation shares its name with the one in a subsequent `x.read()` invocation, but both variables point to different files.

2.2.4 Data Flow Characteristics

The data flow characteristics in a pattern's form concern the range of run-time values software-entities can assume at run-time. They restrict the software entities in the form to entities through which values are denoted (e.g. variable declarations), to entities that evaluate to a value at run-time (e.g. expressions) or to entities that encompass them (e.g. a variable reference within a statement). Machine-verifiable data flow relations range from the relation between the use of a variable and its definition to the relation between a method invocation and its possible target method declarations. Other data-flow relations state whether two expressions point to the same value at run-time. Again, these relations can be required to hold for one, some or all run-time executions of the program under investigation.

Syntactically, the software entities in a data flow relation may differ significantly. This was for instance the case for the `x` and `y` variables in the bug pattern above. Relying on data flow characteristics, a pattern's form does not have to enumerate the multitude of syntactic variations in which relations between run-time values can be established. Often, the entire program must be considered in order to determine that two of its software entities are in a data flow relation.

2.3 Applications of Pattern Detection in Software Engineering

This section briefly highlights some applications of pattern detection tools in software engineering.

2.3.1 Applications in Quality Assurance

For many of the aforementioned software patterns, the overall quality of a program can be ensured by a pattern detection tool that asserts the presence or absence of the pattern's instances in or from the program's source code. Examples include patterns prescribing coding conventions and software engineering best practices or patterns describing bugs and security vulnerabilities. Ideally, such tools are applied *continuously* throughout the program's entire life-cycle such that quality violations are reported the moment they are introduced —either during forward engineering, maintenance or re-engineering of the program.

The full potential of pattern detection in quality assurance is however only realized by tools that support the detection of *user-specified patterns*. Coding conventions and programming styles are often specific to a project development context.

While specialized bug detection tools excel at the detection of common bug patterns such as potential null pointer dereferences, a tool supporting the detection of user-specified patterns can also be applied to application-specific bugs such as violations of the protocol an application programmer interface (API) expects to be adhered to. During development, such tools moreover allow developers to check that a discovered bug is not more widespread throughout the application.

Paraphrasing Engler [ECCH00], "developers understand the semantics of the system operations they build and use but do not have the mechanisms to check or exploit these semantics automatically". Tools restricted to a fixed pattern repository "have the machinery to do so, but their domain ignorance prevents them from exploiting it". Provided developers can communicate the form of the desired patterns, user-specified pattern detection tools can bring machinery and domain-specific semantics together. Applied to higher-level patterns documenting program design, continuous detection of user-specified patterns can also ensure that the design and implementation of the program under investigation remain synchronized (coined co-evolution by Wuyts [Wuy01]).

2.3.2 Applications in Program Comprehension

Developers can apply pattern detection tools in various ways to improve their understanding of an unfamiliar program. Software patterns establish a common vocabulary for software developers. Applied to well-known patterns, pattern detection tools offer a more abstract view of the program under investigation. This view comprises terminology that is of a higher abstraction level than that of the actual software entities in a pattern's form. Each localized instance of the aforementioned enumeration idiom for instance conveys which collection is being enumerated over in the program's source code and which iteration or recursion constructs implement the enumeration. Wills [RW90, Wil92] takes this application of pattern detection to the extreme of automated program recognition: a hierarchical view of the program under investigation is produced comprising recognized pattern instances and the implementation relations among them.

As patterns occur in non-arbitrary contexts, localized pattern instances can moreover convey more than the sum of the entities and relationships their form comprises. This is especially true for design patterns, proven solutions to design problems, of which the form is complemented by a design rationale and discussion of the trade-offs they implement. The view that their localization conveys information about why they are included in the program under investigation is supported by for instance Keller et al. [KSRP99]. Prechelt et al. [PULPT02] also found in empirical experiments that maintainers equipped with explicit design pattern information solve maintenance tasks quicker and with fewer errors.

Recalling that instances of software patterns comprise software entities in particular relationships, tools that support the detection of user-specified patterns can also be applied by a developer to verify the mental model he or she gradually builds about the inner workings of the program under investigation. This model is correct when the tool asserts the presence of the user-specified software patterns it comprises. Such tools therefore support views of program comprehension as an iterative process of mental model construction, verification and revision (see [vMV95] for an introductory overview).

2.3.3 Applications Relied upon by Meta Programming Tools

While the program comprehension and quality assurance applications enumerated above represent direct instances of pattern detection, pattern detection is also implicitly relied upon to varying extents by tools that manipulate programs. Such programs that manipulate other programs are in general referred to as *meta programs*.

This is for instance the case for tools that produce visualisations of (see [GC08] for a recent survey) or metrics about (see e.g. [LM06]) about the program under investigation in which the software entities involved in a visualisation or metric are restricted to those exhibiting specific characteristics. Richner [RD99, Ric02] presents an interesting example where the detection of user-specified, structurally characterized patterns produces custom visualisations.

Refactoring tools, of which the Smalltalk refactoring browser [RBJ97] represents an early example, for instance implement behavior-preserving source-to-source program transformations intended to improve the quality of a program's implementation [Opd92, FBB+99] of which the applicability can be determined by the presence of patterns that fit our liberal definition of a software pattern. This is also the case for the *semantic patches* of the program transformation tool SmPL [PLM06] which describe modifications to multiple program sites intended to automate collateral evolutions, defined as evolutions of a library interface that also affects its clients, in device drivers. SmPL has in fact been applied outside its original domain to the detection of bug patterns [LBH+08]. Pattern detection is involved to similar extents in automatic software migration through program transformation. In contrast to program comprehension and quality assurance applications of pattern detection, it is important in program transformation applications that all reported instances match the pattern's form exactly as these determine the sites where transformations are applied.

A very specific form of pattern detection is also relied upon by programming languages from the aspect-oriented programming paradigm [KLM+97]. These feature pointcut expressions which describe a set of join points in the program under investigation where implementation parts of an aspect and the program can be joined. Intended for the separation of crosscutting concerns, there is more to aspect-oriented programming than the localization of join points. Publications in which expressive pointcuts serve a pure pattern detection purpose are nonetheless testament to the connection (e.g. [SB05] in which bug patterns are detected at run-time).

2.4 Design Dimensions of a Pattern Detection Tool

In this dissertation, we will consider only pattern detection tools that support the detection of user-specified software patterns. Such tools are not limited to a fixed catalogue of patterns. We identify three important dimensions in the design of such a tool: the language in which a pattern's characteristics can be specified, the mechanism the tool employs to detect program elements that match the pattern specification and the representation of the program in which such pattern instances are to be found. The tools referenced throughout this section to exemplify each dimension are discussed in the literature study of Chapter 3.

2.4.1 Pattern Specification Language

The *"pattern specification language"* is the language in which developers express a pattern's form. It is the most prominent point of divergence among tools as it determines which and how machine-verifiable pattern characteristics can be specified. Following the categorization of programming languages in paradigms, we distinguish imperative specification languages from declarative specification languages.

Imperative specification languages Imperative specification languages consist of an imperative programming language and an application programmer interface (API) through which the tool can be algorithmically instructed to search for an application-specific pattern. Many tools support custom traversals of the program's source code (e.g. PMD [PMD08], CHECKSTYLE [Che08] and SPOON [PNP06]) or byte code (e.g. FINDBUGS [HP04]) by extending an implementation of the Visitor design pattern. The resulting program cannot be considered a specification of a pattern's characteristics, but rather one of the possible implementations for its search. While such algorithmic implementations of the search process lend themselves to pattern-specific optimizations, the knowledge required about a tool's internals is often on par with that of its implementers. As a similar search process is shared by multiple patterns, parts of the implementation will moreover have to be repeated.

Declarative specification languages Tools offering a declarative specification language do not require an algorithmic implementation of the search for a particular pattern, but identify program elements using a built-in detection mechanism. Generalizations of regular expressions (e.g. over abstract syntax trees as used by TAWK [GAM96] or over control flow graphs as used by Liu et al. [LRY+04] and JUNGL [VEdM06]), temporal logic formulae (e.g. over control flow graphs as used by TRANS [LdM01]) and even source code fragments (e.g. Sellink et al. 's Native Patterns [SV98]) are all instances of declarative pattern specifications encountered in the literature. Some of the programs written in a general-purpose declarative programming language can be considered declarative specifications of a pattern's characteristics as well.

At first sight, tools using declarative programs for the specification of patterns are akin to tools using imperative programs for the implementation of the pattern search process. However, a common characteristic of declarative programming languages sets both approaches apart. In imperative programming languages, programmers specify exactly how the solution to a problem is to be found using step-by-step algorithmic descriptions. In contrast, declarative programming languages allow the problem itself to be specified. The programming language will find a solution on its own, relying on a specific problem solving strategy embodied by the language's operational semantics. In a pattern detection setting, this property of declarative programs facilitates their use as specifications of a pattern's characteristics rather than its search process. Torgersson [Tor96] gives an introduction to the declarative paradigm and the diverse general-purpose programming languages it comprises.

2.4.2 Pattern Detection Mechanism

Given a declarative specification of a pattern's form, it is up to the *"pattern detection mechanism"* to localize matching software entities in the program under investiga-

tion. Depending on the actual format the declarative pattern specification takes, the employed detection mechanisms range from matching the program's AST with the one of a pattern specified as a source code fragment (e.g. IntelliJ's *Structural Search and Replace* feature [Mos05]) to constraint solving for patterns specified as a set of constraints (e.g. PTIDEJ [Gué03]).

The choice for a particular declarative specification language does not entail a unique pattern detection mechanism. A tool's pattern specification language and its detection mechanism are separate dimensions in the design space. Multiple algorithms with different properties can for instance be devised to localize software entities that are executed in the order dictated by a machine-verifiable control flow characteristic (e.g. both de Moor et al. [dLW03] and Liu et al. [LRY+04] present an algorithm to check that sequences of executed instructions fit a form). We will refer to the algorithms employed by the detection mechanism in the search for matching software entities as *"search strategies"*.

The same even goes for specifications taking the form of a program in a declarative programming language. Datalog [CGT89] programs can for instance be evaluated according to various strategies. While we argued above that general-purpose declarative programming languages are suited for the specification of patterns, both domain-specific extensions to their regular syntax (e.g. JTL [CGM06b]) and their regular problem solving strategy (e.g. ASTLOG [Cre97]) are common. These respectively intend to increase the expressiveness of the resulting specifications and render the localization of pattern instances more effective. After all, the declarative nature of the programming language does not impede the user from implementing algorithmic searches for a pattern.

The search strategy is not the only point on which detection mechanisms tend to differ. Another point of variation encountered in the survey of the state of the art (see Chapter 3) is whether or not users are able to steer the devised search strategies. Such provisions range from simple declarations external to the specification that influence whether all type-related syntactic characteristics in the specification are to be adhered to strictly, to advanced provisions for interactivity that support soliciting the user for advice on intermediate results. Another variation point concerns whether or not the detection mechanism provides a ranking for the instances of a pattern it reports.

2.4.3 Program Representation

To localize instances of a specified pattern, the detection mechanism needs to be able to examine the program under investigation as data. To this end, it requires a representation of the program under investigation. The kind of information about the program this representation carries can be considered a key dimension in the design of a general-purpose pattern detection tool. It shapes the kind of patterns the tool is able to detect, the effort that is required to specify a pattern's characteristics with respect to the provided information and the precision and recall ratios the detection mechanism is able to attain. This is clarified below:

- The information carried by this representation first of all determines the kind of software patterns the tool is able to localize. In case this information is purely structural in nature, it will be impossible to apply the tool to the detection of pattern forms that include other characteristics. When a representation of a Java program for instance lacks information about method bodies,

localizing instances of the potential null dereference bug pattern will be impossible without additional information.

- Moreover, the information carried by the program representation determines the effort that is required from the user to specify a pattern's form and the effort that is required from the detection mechanism to localize its instances.

  Pattern specification is straightforward when a pattern's characteristics can be expressed through direct references to the information that is explicitly carried by the program representation. For instance, in a representation that explicitly indicates which pairs of instructions are executed consecutively, an existential quantification over this information suffices to specify a pattern of two consecutively executed instructions.

  The detection mechanism realizes the semantics of the specification language. Any information not carried by the program representation that is necessary to verify that a software entity exhibits a specified characteristic will have to be derived by the detection mechanism itself. The detection mechanism bridges the gap between the information the program representation carries and the pattern characteristics that can be expressed in the specification language. For instance, information about the execution order of instructions can be derived from abstract syntax trees (see Section 2.5.1) or extracted from control flow graphs (see Section 2.5.3). However, from the point of view of the user, this situation does not differ from the one described above. In both cases, the user can concentrate on specifying the pattern rather than deriving any information its detection requires.

  Considerably more effort is required from the user in case the specification language does not support the expression of certain pattern characteristics. Where possible, users can express the unsupported characteristics in terms of those that are supported. For instance, most characteristics can be expressed in terms of syntactic characteristics albeit not straightforwardly. As a final resort, users can attempt to express the unsupported characteristics by quantifying over program information they derive themselves —provided the necessary information can be derived from the carried information and provided the tool's specification language is sufficiently powerful to specify how this derivation is to proceed.

- Finally, the program representation has a non-negligible influence on the effort that is required to minimize the number of program entities that are identified mistakenly as instances of the specified pattern (i.e. false positives). The same goes for the effort required to maximize the tool's *recall* (i.e. the ratio of the true positives to the sum of the true positives and false negatives the tool reported [Hea78]). The tool's *precision* is measured in terms of the ratio of true positives to the sum of all true and false positives. For instance, associating line numbers with AST nodes gives an indication of the order in which each node is executed. This representation is simple, but might lead to a lower recall in the search for two consecutively executed instructions. According to this representation, the last statement in a loop is never executed before the first statement in the loop. False positives are also reported for the same pattern. According to the representation, statements in the `else` branch of an `if`-statement are executed after the statements in

the `then` branch. Depending on the tool's intended application area, these defects might however be acceptable to users favoring detection speed.

The information in a program representation can be obtained from the program under investigation through various techniques. This is especially true for information about the program's run-time behavior. For object-oriented programs this information can take different forms ranging from the execution order of statements in a method body to a whole-program method invocation graph. Behavioral information can be obtained either by monitoring the program's behavior at run-time or by predicting it through an analysis of its code at compile-time. These program analyses are called *dynamic analysis* and *static analysis* respectively.

Dynamic analysis While the behavioral information offered by a *dynamic analysis* is very precise, it is only valid for one of many possible program executions. Different behavior might be observed in a subsequent execution and certain parts of the program might not be executed at all. Although a pattern instance might be identified with great precision, its presence in or absence from all possible program executions can therefore not be established universally.

Dynamic analysis furthermore requires a high degree of user involvement. The program must be executed along a well-defined scenario that either concentrates on particular parts of the program or exercises as much of the program's code as possible. Determining a suitable execution scenario might require detailed knowledge of the program.

Finally, while a regular program execution might already take a substantial amount of time, the overhead introduced by run-time monitoring aggravates this problem. Depending on whether pattern detection is interleaved with program monitoring or is performed after the program has run, a substantial amount of memory might be required to store the required behavioral information.

We refer the reader for a more elaborate discussion on the drawbacks and advantages of dynamic analysis in pattern detection to [DGD05, De 04, DMG+06, MDB+06].

Static analysis The focus of this dissertation is on pattern detection using program representations that can be obtained without executing the program under investigation. Any behavioral information in such a program representation must result from a *static analysis* [NNH05] which exploits the semantics of the base program's programming language to derive facts about its behavior.

While the behavioral information derived by a static analysis is valid for all possible program executions, it approximates the application's actual behavior. In this sense, both static analysis and dynamic analysis techniques complement each other for pattern detection —see for instance [Ern03].

Some of the tools included in the survey of the state of the art in Chapter 3 therefore augment their statically obtained information with an optional dynamic analysis to refine their results. Costly program monitoring can then be restricted to those parts of the program the static analysis couldn't provide precise enough results about.

```
1 public void insertElement(Object x) {
2   Iterator i = list.iterator();
3   while (i.hasNext()) {
4     Object o = i.next();
5     addObjectToCollection(x, (Collection) this.self().list);
6   }
7 }
```

Figure 2.1: Concrete syntax of the Java method `insertElement(Object)`.

2.5 Supporting Machine-Verifiable Pattern Characteristics

In this section, we discuss for each kind of pattern characteristic which combinations of a specification language, detection mechanism and a statically obtained program representation support its detection. We limit ourselves to those combinations encountered in the pattern detection literature. Section 2.5.1, Section 2.5.2, Section 2.5.3 and Section 2.5.4 discuss support for the syntactic, structural, control flow and data flow characteristics introduced in Section 2.2 respectively.

A general-purpose pattern detection tool must support each of these characteristics explicitly. Otherwise, users are forced to implement the missing support for a necessary characteristic themselves. We will show for each characteristic why this is hard. We will also show that this might lead to false positives being reported and a lower recall of pattern instances.

2.5.1 Supporting Syntactic Characteristics

Syntactic characteristics are introduced in Section 2.2.1. They require an appropriate specification language, detection mechanism and program representation. The support provided by most integrated development environments, locating strings in the program's source code that match a specified regular expression, is not adequate. It is sensitive to the textual organisation of the program. The possibility of white-space variations and spurious characters must be accounted for in specifications. With most regular expression languages lacking support for meta-variables or abstraction facilities, this is hard. Paul [PP94] illustrates this for the pattern consisting of an `int` and `char` variable declaration. Its specification as a regular expression is given below. Note how white-space and all permutations of the two declarations must be accounted for. The possibility of pattern instances that span multiple lines is not taken into account. Such instances will not be recalled.

```
1 (.*int[ ]*x[ ]*;.*char[ ]*c[ ]*;) | (.*char[ ]*c[ ]*;.*int[ ]*x[ ]*;)
```

A representation of the program that abstracts away from its textual organisation is therefore highly desirable. Ideally, the program's text adheres to the *concrete syntax* of the programming language it is written in, which is usually described formally as a context-free grammar. Rather than relying on concrete syntax trees that encode the program's syntactic structure with respect to this grammar, most program manipulating tools rely on *abstract syntax trees* (AST). Concrete syntax features such as parentheses are often absent from abstract syntax trees as their hierarchical structure suffices for instance to encode expression evaluation precedence.

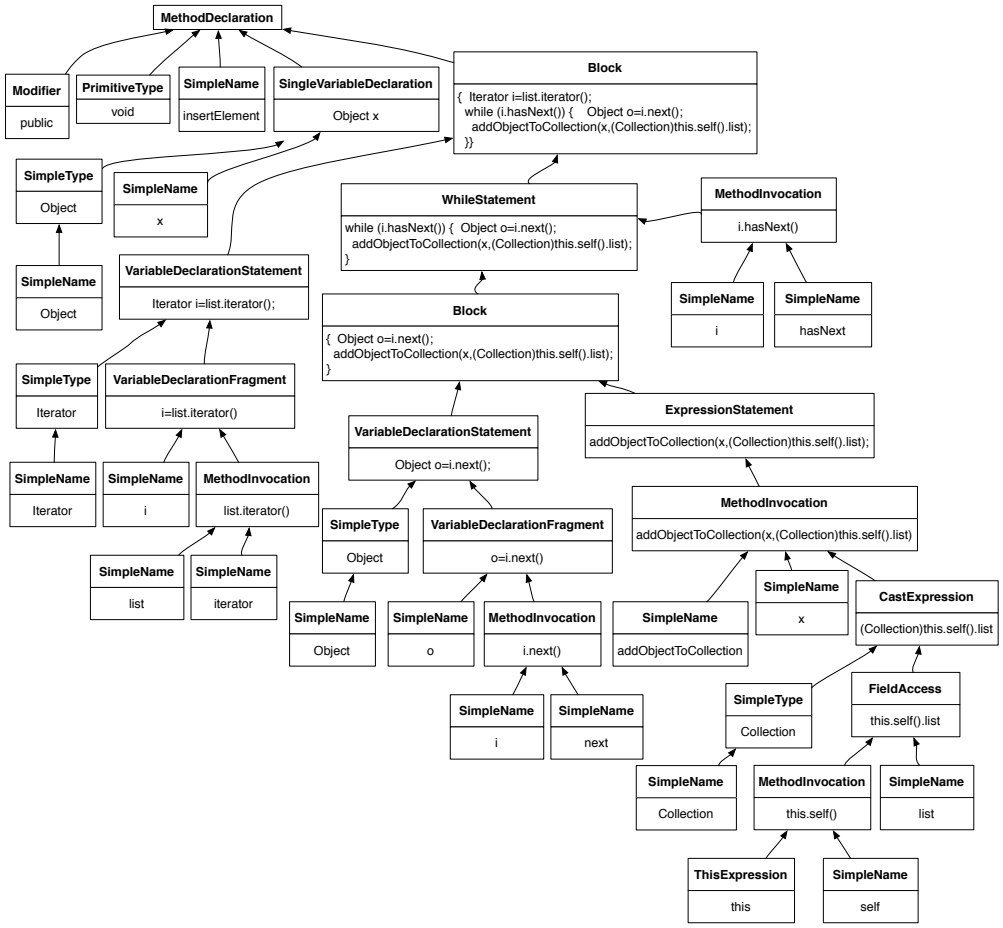


Figure 2.2: Graphical representation of an abstract syntax tree for the Java method `insertElement(Object)`, depicted in Figure 2.1, obtained through the DOM component of the Eclipse Java Development Toolkit (JDT).

Figure 2.2 depicts a graphical representation of an AST for the Java method `insertElement(Object)` of which the concrete syntax is depicted in Figure 2.1. This particular AST is produced by the DOM component of the Eclipse Java Development Toolkit (JDT) [Ecl08a] and is relied upon by all Eclipse plug-ins that manipulate Java source code. Its implementation is object-oriented. Each AST node is depicted along with its concrete syntax representation. We will revisit this component in the next section.

Section A.1 details how abstract syntax trees can be obtained for the program under investigation. Some pattern detection tools (e.g. SCRUPLE [Pau92, PP94]) apply a further *canonicalization* to the obtained AST. Syntactically differing, but semantically similar constructs are mapped into a canonical form. It obviates the need to specify all semantically similar syntactic constructs in a pattern's form. However, in order to avoid confusing the user, software entities from the original rather than the canonicalized source code should be reported.

CONDATE [Vol06a, Vol06b] and MJ [BE03] base their program representation on a control flow graph (see Section 2.5.3) rather than an AST. The nodes in these graphs stem from an intermediate representation and a bytecode representation respectively. The tools support syntactic characteristics by reconstructing the program's AST from the control flow graphs. However, this is not straightforward. There is no one-to-one mapping for most programming languages. Moreover, not all characteristics can be supported. AST nodes with internal control flow such as iterative loop constructs are missing from these graphs as they are expanded in their atomic operations.

Apart from a program representation, tools supporting syntactic characteristics still require a suitable specification language and associated detection mechanism. For instance, TAWK [GAM96] offers a generalization of regular expressions over AST nodes. ASTLOG [Cre97] offers a Prolog specialized in AST traversals. Ideally, the specification language features abstraction facilities that enable partial reuse of specifications. However, these should not be used to express non-syntactic characteristics in terms of syntactic characteristics. This results in convoluted specifications with recurring parts that might lead to a lower recall and false positives. For instance, expressed in terms of syntactic characteristics, the specification of a pattern of two consecutive function calls must account for the situation in which both occur at different levels of nesting in a procedure. We will revisit this statement in our discussion of the other characteristics.

2.5.2 Supporting Structural Characteristics

Structural characteristics are introduced in Section 2.2.2. The actual nature and granularity of the program information that supports them differs greatly —even for base programs in the same programming language. Some representations of object-oriented programs offer only coarse-grained information about a program's classes and their inheritance relations (e.g. the one used by PAT [KP96] which is extracted from the header files of a C++ program). Others offer information about the program's methods encompassing an enumeration of the fields they read and write (e.g. the ones used by jQuery [De 06, JD03] and CODEQUEST [HVd06]). However, common to all structural information is that their granularity prohibits a reconstruction of the source code that results in a completely functional program.

The motivation to support structural characteristics explicitly rather than have the user specify them through syntactic characteristics is two-fold:

- Expressing structural characteristics in terms of syntactic characteristics is not straightforward. It results in large expressions with idiomatic recurring parts because structural entities and relations must be derived from the complete syntactic structure of the program's source code. Consider the *Overrider* pattern, introduced in Section 2.1, which describes a class that exclusively consists of overriding methods. Without explicit support for the overriding and inheritance relations, the pattern must be expressed in terms of multiple AST traversals.

- Specified explicitly, the detection mechanism can profit from the reduction in search space that the use of structural information implies. With respect to an AST of the whole program, fewer software entities must be considered. However, these tools often pay for the reduction in search space with false positives. Many design patterns are structurally equivalent. Instances of the

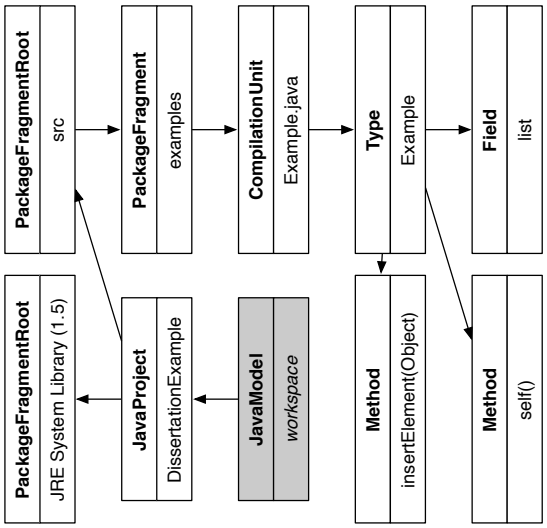


Figure 2.3: Graphical representation of a structural representation of the Java program in which the method `insertElement(Object)`, depicted in Figure 2.1, resides as obtained through the Model component of the Eclipse Java Development Toolkit (JDT) .

Strategy and *State* design patterns can for instance only be distinguished by analysing the program's source code [NSW+02].

Figure 2.3 depicts a graphical representation of a structural representation of the Java program in which the method `insertElement(Object)`, depicted in Figure 2.1, resides. It is obtained through the Java Model component of the Eclipse Java Development Toolkit (JDT) [Ecl08a]. The class of which structural elements are an instance is depicted at the top of the node.

Section A.2 details how structural information can be obtained for the program under investigation. Most AST-based tools incorporate structural information as an index into their abstract syntax trees, e.g. :

- The Eclipse [Ecl08b] integrated development environment is an illustrating example. Its Java Development Toolkit (JDT) comprises an API against which plugins that manipulate Java code can be developed [ABL05]. Plug-ins can access the code in the IDE through two distinct interfaces that can be characterized as an abstract syntax tree and a structural program representation respectively: the DOM and Java Model components of the JDT. The former offers access to all of a program's syntactic constructs the latter only includes summary information about its structural entities such as packages, compilation units, classes, field and method signatures. Its lightweight nature facilitates keeping the structural representation in sync with the program's code and therefore functions as basis for many navigational and program query tools. Tools that have to manipulate the program's code rely on the abstract syntax tree representation instead.

- Every Smalltalk program has access to a program representation that combines syntactic and structural information. The reflective capabilities of the

```
1  public void insertElement(java.lang.Object)    {
2    examples.Example r0, $r5;
3    java.lang.Object r1;
4    java.util.Iterator r2;
5    java.util.List $r4, $r6;
6    boolean $z0;

7    r0 := @this: examples.Example;
8    r1 := @parameter0: java.lang.Object;
9    $r4 = r0.<examples.Example: java.util.List list>;
10   r2 = interfaceinvoke $r4.<java.util.List: java.util.Iterator iterator()>();
11   goto label1;

12   label0:
13   interfaceinvoke r2.<java.util.Iterator: java.lang.Object next()>();
14   $r5 = virtualinvoke r0.<examples.Example: examples.Example self()>();
15   $r6 = $r5.<examples.Example: java.util.List list>;
16   virtualinvoke r0.<examples.Example: void addObjectToCollection(java.lang.Object,
17                                          java.util.Collection)>(r1, $r6);

18   label1:
19   $z0 = interfaceinvoke r2.<java.util.Iterator: boolean hasNext()>();
20   if $z0 != 0 goto label0;

21   return;
22 }
```

Figure 2.4: Concrete syntax of the JIMPLE intermediate representation for the Java method `insertElement(Object)`, depicted in Figure 2.1, obtained through the SOOT Java Optimization Framework.

> Smalltalk programming language [Riv96, GR83] allow a program to inspect the classes and methods it is comprised of. In addition to this structural self-representation, programs can access the abstract syntax trees of their methods at run-time through the Smalltalk compiler or the parser of the Refactoring Browser [RBJ97] which is included in most integrated development environments.

We will revisit the above representations in the discussion of the logic programming language SOUL in Chapter 5. The logic predicates in its LICOR library rely on linguistic symbiosis [GWDD06] to access the reflective protocols of Smalltalk —thus enabling pattern detection for Smalltalk. The Eclipse-delivered representations will be revisited in that chapter too as one of our own extensions to SOUL comprises the logic predicates in the CAVA library which access the code in an Eclipse workspace through a symbiosis between Java and Smalltalk —thus enabling pattern detection for Java.

2.5.3 Supporting Control Flow Characteristics

Control flow characteristics are introduced in Section 2.2.3. To support them, pattern detection tools need information about the order in which instructions are executed at run-time. Given a sufficiently powerful specification language, users can derive this information themselves. However, as exemplified in Section 2.4.3, expressing control flow characteristics correctly in terms of syntactic characteris-

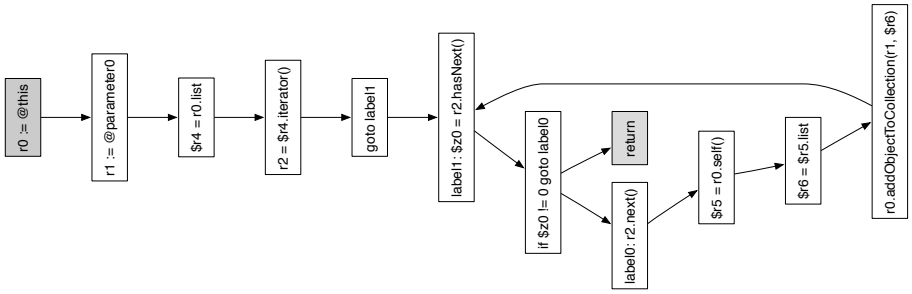


Figure 2.5: Graphical representation of a control flow graph for the Java method `insertElement(Object)`, depicted in Figure 2.1, obtained through the SOOT Java Optimization Framework with nodes in the graph originating from its JIMPLE intermediate representation depicted in Figure 2.4.

tics is hard. The semantics of the programming language must be taken into account. The execution order of instructions may vary with each execution of the program. Without information about the run-time values of expressions, control constructs must be handled conservatively. For instance, the conditional expression of an `if`-statement can be followed by either branch of the statement. Any errors made by users in this derivation might lead to false positives and a lower recall. Without support for a pattern's control flow characteristics, its specification grows with idiomatic recurring parts that are not specific to the pattern itself.

Explicit support for control flow characteristics is therefore desirable. Specification languages encountered in the literature range from formulae in temporal logic (e.g. TRANS [LdM01]), over generalizations of regular expressions specifying sequences of executed instructions (e.g. Liu et al. [LRY+04] and JUNGL [VEdM06]), to expressions specifying that a specified instruction is executed eventually after another specified instruction without a third specified instruction ever being executed in between (e.g. Condate [Vol06a, Vol06b]).

Representing Control Flow

Analogous to an AST that abstracts away from the program's textual organisation, a control flow graph abstracts away from its syntactic organisation. This is desirable as consecutively executed instructions are often syntactically dispersed. Nodes in the directed graph represent instructions. Edges in the graph connect instructions that might be executed consecutively at run-time. A node has multiple predecessors when branches from the program's control flow join. Conversely, a node has multiple successors when the program's control flow splits. A control flow graph may have cycles. Most cycles originate from back-edges that connect the last statement in the body of an iterative control statement with its conditional expression. Like all instructions with internal control flow, such statements are usually expanded into the atomic operations they comprise.

Figure 2.5 depicts a graphical representation of a control flow graph for the Java method `insertElement(Object)`, depicted in Figure 2.1. It is obtained through the SOOT Java Optimization Framework [VRCG+99]. The nodes in the graph stem from its JIMPLE intermediate representation depicted in Figure 2.4. It will be revisited in Section 5.3.4.

Control flow graphs are the starting point for many traditional program analyses [NNH05]. For instance, an analysis that computes which variable assignments reach a variable reference. Section 2.5.4 discusses how their results support data flow characteristics. In the bug pattern detection literature, an edge-labelled control flow graph is more common (e.g. [OO90] representing an early instance). The edges of the graph rather than its nodes carry program instruction labels. Each edge is labelled by its target instruction. This has the advantage that information about whether the true or false branch is taken out of a conditional expression can be encoded easily.

The control flow graph for a procedure can be constructed inductively by connecting the sub-graphs of the statements it comprises. A whole-program control flow graph can be obtained by cloning the graph of the called procedure at each call site, but this is not space effective. The alternative is to link procedure calls to the entry point of the callee's graph and to link its exit points back to the caller. The construction of whole-program control flow graphs for higher-order and object-oriented programs can no longer proceed inductively. Complications and possible solutions are sketched in Section A.3. There, we also sketch the construction in the presence of reflection. Because of these complications, it is better not to burden the user with the construction of a control flow graph.

Extracting Execution Order Information

A control flow graph is a compact representation of the possible executions of a program. However, a simple quantification over such a graph does not suffice to detect control flow characteristics. The paths through this graph must be computed. Paths represent sequences of consecutively executed instructions. Every path from its entry to one of its exit nodes corresponds to a trace of instructions that might be observed at run-time. Again, it is best not to burden the user with the computation of these paths. Cycles in the graph are an example of a complication that users must handle otherwise. *Infeasible paths* and *unrealizable paths* are other complications that might introduce false positives. The detection mechanisms encountered in the literature vary on whether and how these complications are dealt with.

Infeasible paths can be eliminated by tracking the truth value of conditional expressions. For instance, the search strategy maintains that an expression must evaluate to true on paths where it follows the true-branch out of this expression. If it encounters an equivalent or derived expression later on, it must not follow the false-branch out of this expression. However, accurately maintaining data dependencies from the encountered equalities is hard. According to our literature study, it is only performed by METAL [ECCH00].

In order to recognize instances of a sequencing pattern that crosses function boundaries, an *inter-procedural* search strategy (e.g. the one employed by METAL [ECCH00]) is necessary. It joins the graphs of individual functions. Search strategies that are restricted to instances within a single function are called *intra-procedural* (employed by most of the other tools specialized in control flow characteristics). However, multiple call sites result in control flow splits at the exit points of callees for link-based whole-program graphs. The search strategy has to take care not to follow *unrealizable paths* by ensuring that the successors of a function's exit nodes agree with the function call earlier on the path.

We conclude that both the derivation of an accurate control flow graph and the

```
1  public void addObjectToCollection(java.lang.Object, java.util.Collection) {
2    examples.Example r0;
3    java.lang.Object r1;
4    java.util.Collection r2;

5    r0 := @this: examples.Example;
6    r1 := @parameter0: java.lang.Object;
7    r2 := @parameter1: java.util.Collection;
8    interfaceinvoke r2.<java.util.Collection: boolean add(java.lang.Object)>(r1);
9    return;
10  }

11 P(r2) = {
12 AllocNode 3986 new java.util.Vector in <examples.Example: void anotherInvoker()>,
13 AllocNode 3983 new java.util.LinkedList in <examples.Example: void initializeContainer()> }

14 P(r6) = {
15 AllocNode 3983 new java.util.LinkedList in <examples.Example: void initializeContainer()> }
```

Figure 2.6: Concrete syntax of the JIMPLE intermediate representation for the Java method `addObjectToCollection()`, invoked by the `insertElement(Object)` method depicted in Figure 2.1, and the points-to sets for their may-aliasing local variables `r2` and `r6` respectively as obtained through the SPARK [Lho02] context-insensitive points-to analysis of the SOOT Java Optimization Framework.

efficient localization of instruction sequences on its paths is hard. It is better not to burden the user with either and to support control flow characteristics explicitly.

2.5.4 Supporting Data Flow Characteristics

Data flow characteristics are introduced in Section 2.2.4. In theory, data flow characteristics can be expressed in terms of syntactic characteristics. However, this is not trivial. The semantics of the programming language must be taken into account. Often, the entire program must be considered in order to determine that two of its entities are in a data flow relation. For instance, a field can be assigned at many different locations. Moreover, fix-point computations may be necessary in the presence of loops.

Data flow information pertains to the range of possible run-time values expressions can assume, as well as their origin and how they are related. Computed at compile-time, data flow information is valid for all possible executions of the program under investigation. It is crucial to the successful application of program optimizations in modern compilers —offering answers to important problems such as the scope of definitions and which expressions may, must or definitely do not point to the same memory location. *Reaching definitions* and *points-to analysis* [Hin01], the data flow analyses [NNH05] that solve said problems, are also crucial in pattern detection. In this context, their results support data flow characteristics. Specifications do not have to enumerate the multitude of syntactically differing program snippets that establish the same data flow relation between variables. For instance, an aliasing relation can be established between two variables by assigning the value of one variable to the other or by assigning to one variable the value of a third variable that already aliased the other.

Figure 2.6 depicts a textual representation of the points-to sets for the

may-aliasing local variables `r2` and `r6` stemming from the intermediate representation of the method `addObjectToCollection()` and method `insertElement(Object)` respectively. The latter is depicted in Figure 2.4. The points-to sets are obtained through the SPARK [Lho02] context-insensitive points-to analysis of the SOOT Java Optimization Framework. We will revisit this analysis in the remainder of this section.

Section A.4 lists some existing implementations of data flow analyses, the results of which can be incorporated in the program representation of a pattern detection tool. Data flow analyses differ greatly in the kind of information they compute and the conclusions their results allow to be drawn about the program's run-time values. In the remainder of this section, we will illustrate the complications that arise in accessing and interpreting the results of certain analyses correctly. Data flow representations must therefore be accompanied by a pattern specification language as well as a detection mechanism that hides and interprets the analysis-specific nature of their information respectively. The variety among data flow analyses cannot only give rise to completely different search strategies for the same pattern, but also to a different recall and precision among pattern detection tools that employ their results as program representation.

Complications Arising from the Use of Data Flow Information

Informally, many static analyses can be regarded as if they were executing the analysed program with abstract descriptions of the concrete values that appear during an ordinary program execution. This is especially true for those analyses constructed according to the abstract interpretation methodology [CC77]. To make this idea somewhat more tangible, consider the sign of an integer variable which can be used as an abstract description of the set of values it can assume at run-time. This abstraction constitutes a source of imprecision of which the clients of such a sign analysis must be aware of. Likewise, the manner in which the execution of the program is simulated constitutes a source of unknowns.

Unknowns in data flow information In order to be useful, the information derived by a data flow analysis must agree with the program's actual run-time values. For an integer sign analysis, this for instance entails that all identified absolutely negative expressions do evaluate to a negative integer (< 0) at run-time. However, not all of a program's negative expressions might be identified. For integer expressions residing in reflectively invoked code, a simple analysis might for instance offer no sign information at all ($\perp$). Unknowns are therefore to be expected in the results.

Imprecision in data flow information It is not exceptional for results to be imprecise, as the deliberate introduction of less precise, but valid information is key to the manner in which they are derived:

- The actual abstraction employed to describe possibly unbounded sets of concrete values constitutes a first source of imprecision. While an expression's value might in reality be restricted to odd numbers only, their abstraction in the aforementioned sign analysis represents a larger set of concrete program values which also includes even numbers. Moreover, as the program's execution is simulated with these abstract values, they are affected by

the program's instructions. While the effect of multiplying a positive and negative integer is well-known, their addition can result in a positive as well as a negative sum. The analysis can therefore not be conclusive.

- Finally, as the computed information should be valid for all of the program's executions, the analysis is forced to introduce more imprecision to account for all the paths through its control flow graph. For instance, both branches of a conditional contribute to the analysis results for a variable even though the variable is assigned completely different values in each branch. In order to agree with both control flow paths that are possible, the unified results for the variable therefore include weaker information than is actually true on either path. In case the variable's value is deemed to be negative (< 0) in one branch and zero in the other, the analysis can only conclude that the variable's value after the conditional must be non-positive (≤ 0). This is a conservative, yet reasonably precise approximation. When both branches yield contradicting results, the overall result indicates that any sign is possible ($\top$).

The most conservative conclusion has to be avoided as much as possible for an analysis to be useful. There is a trade-off to be made between precision and analysis time. Abstract value descriptions and the merging of information from distinct paths respectively deal with sets of potential run-time values and execution states.

Accessing and drawing conclusions from data flow information In order to interpret data flow information correctly, users must be aware of the conclusions this information allows to be drawn about the program's run-time values. In order to access their results, some of the more precise analyses require knowledge about the execution state abstractions that are employed.

Accessing results Little impedes the results from the classical data flow analyses used in compiler construction from being queried. Most implementations annotate the entities in their underlying program representation or provide an index into the analysis results. However, problems arise when this representation does not correspond to the one employed by the pattern detection tool. Most data flow analyses rely on a control flow or intermediate representation derived from the program's compiled code.

Accessing context-sensitive results Data flow analyses implement different trade-offs with respect to precision and cost. For instance, at the cost of some precision, *flow-insensitive* inter-procedural analyses save time and space by only propagating analysis information along the program's procedure call and return edges while ignoring execution order inside individual procedures. What trade-off is appropriate depends on the needs of the client. Relevant to our discussion is that one of these trade-offs, whether or not the inter-procedural analysis is *context-sensitive*, also impacts the way its results are accessed. As the effect of a procedure might differ per call, a context-sensitive analysis aims to increase precision by computing separate results for each calling context it can discern. Their *context-insensitive* counterparts merge the effects of individual calls thus producing information that fits all of the procedure's calls. From the user's perspective, context-sensitive analyses parametrize their results by a static representation of the program's calling contexts at run-time. In order to

access context-specific results, users need to be able to reconstruct these context representations —usually an abstraction of the topmost entries on the program's call stack.

That this is not always trivial is perhaps best illustrated by the aforementioned *points-to analyses* for Java which compute the set of all heap objects a reference expression may point to at run-time. Here, clients need to be aware of both the heap referee (i.e. the object being referred to) and heap reference (e.g. a variable) abstractions employed by the context-sensitive variants:

Context-sensitive heap references A context-sensitive points-to analyses computes for each reference a points-to set parametrized by an invocation context for the method the reference resides in. While the call sites of the topmost invocations on the call stack usually comprise the context abstractions for procedural languages, in an object-oriented setting the receivers of the topmost invocations on the call stack [MRR02] are more appropriate with respect to scalability and precision [LH06]. Again, such an analysis employs a static abstraction of the concrete objects this receiver may comprise at run-time. As clients need to reconstruct invocation contexts in order to access analysis results, clients must also reconstruct the heap referee abstractions that are part of the heap reference abstractions employed by such an analysis.

Context-sensitive heap referees As points-to sets are computed at compile-time, they comprise abstractions of the concrete objects that populate a program's heap at run-time. In order to ensure that the analysis terminates for programs that create infinitely many concrete objects, its static heap referee abstraction may only comprise a finite amount of abstract objects. A single abstract object must therefore correspond to multiple concrete objects. Analogous to the context abstractions above, a common heap referee abstraction scheme is to represent all concrete objects by their allocation site in the underlying program representation. A context-sensitive referee abstraction in addition records information about the context in which each object is allocated. This helps the analysis to discern objects created at the same site —for instance internal objects created in constructors of data structures [LH06].

Interpreting analysis results Finally, users must know the limits of the calculated results. Again, we will illustrate this using a points-to analysis. Each object a reference points to at run-time must be included in its points-to set. A reference can therefore never point to any object that is not included in its points-to set. Due to over-approximation, the points-to set might however include objects the reference never points to in reality. Clients are therefore able to derive whether two references *may-alias* by checking whether the intersection of their respective points-to sets is not empty. From non-overlapping points-to sets, clients can moreover derive that the corresponding references *must-not-alias*. They can however never derive a *must-alias* relation from the points-to analysis results.

Criterion	*Description*
CSL1	Supports the specification of behavioral and non-behavioral characteristics in a uniform language
CSL2	Results in descriptive pattern specifications
CSL3	Supports expressing explicit points of variation among pattern instances
CSL4	Provides means for abstraction and reuse among specifications
CSL5	Hides program representation details
CDM1	Reports elements from the program's source code
CDM2	Facilitates user assessment of reported instances
CDM3	Supports implicit points of variation among pattern instances
CDM4	Can be extended with user-defined search strategies
CPR1	Includes behavioral and non-behavioral program information explicitly

Table 2.1: Overview of the criteria for a general-purpose pattern detection tool. The acronyms **CSL#**,**CDM#** and **CPR#** stand for **CRITERION FOR THE SPECIFICATION LANGUAGE**, **CRITERION FOR THE DETECTION MECHANISM** and **CRITERION FOR THE PROGRAM REPRESENTATION** respectively.

2.6 Criteria for a General-Purpose Pattern Detection Tool

A *general-purpose* pattern detection tool is not specialized in one particular application of pattern detection, but can be applied equally well to any of the software engineering problems enumerated in Section 2.3. Such a tool obviates the need for an assortment of tools each specialized in one kind of pattern. As a result, users only need to know the specification language of a single tool to benefit from pattern detection.

When fulfilled, the criteria formulated in this section ensure that these properties of a general-purpose pattern detection tool are satisfied. Table 2.1 enumerates all criteria. The criteria are organized according to the previously identified dimensions in the design of a pattern detection tool: the offered specification language (criteria **CSL#**), the employed detection mechanism (criteria **CDM#**) and the program representation relied upon (criteria **CPR#**) .

2.6.1 Criteria for the Pattern Specification Language

CSL1: Supports the specification of behavioral and non-behavioral characteristics in a uniform language

Patterns embodying proven architectural blueprints, coding conventions, best practices, prescribed protocols of application programmer interfaces and application-specific bugs are but a few of the patterns that a general-purpose tool should support. The specification language of a general-purpose tool must support expressing their machine-verifiable characteristics. Examples of software patterns are given in Section 2.1. Their structural, syntactic, control flow and data flow characteristics are discussed in Section 2.2.

In the interest of the tool's accessibility to application programmers, it is important that all machine-verifiable characteristics can be expressed in one *uniform* specification language. As illustrated by the running example in Section 2.2, patterns need not be characterized by a single kind of characteristic.

For those patterns, using a general-purpose tool has no advantage over using a combination of special-purpose tools *if* the specification language is an amalgamation of languages individually designed to express one kind of characteristic. In both cases, users have to know multiple specification languages.

CSL2: Results in descriptive pattern specifications The specification language should result in a *descriptive* specification of the characteristics of a pattern, not in an operational implementation of the search for program elements exhibiting these characteristics. Both ought to be separated. As elaborated on in Section 2.4.1, the latter is best left to the implementers of the tool. This relieves users from having to re-implement a search process for similar patterns. Additionally, such implementations often require a level of knowledge about the tool's internals that is on par with that of its implementers.

To separate pattern specifications from the search for their instances, it is up to the detection mechanism to devise the operational search corresponding to a given specification. Realizing its semantics, the detection mechanism is closely related to the specification language. We will discuss the criteria for the detection mechanism separately.

CSL3: Supports expressing* explicit *points of variation among pattern instances Recalling the definition of a pattern from Riehle et al. [RZ96] (see Section 2.1) as an abstraction of concrete forms that recur in non-arbitrary contexts of use, we distinguish explicit points of variation from implicit points of variation among pattern instances. *Implicit* points of variation among instances represent different implementations of the same characteristic. For example, instances can refer to a type by its fully qualified name or its simple name. These will be revisited in the discussion of the criteria for the detection mechanism as the user should not have to enumerate all implicit variation points in a specification. *Explicit* points of variation among instances represent variations in the characteristics shared by all pattern instances. For example, the field protected by a getter method differs in each instantiation of this best practice pattern [Bec96]. The specification language should support expressing such points of variation in pattern specifications.

It should also be possible to express constraints on explicit variation points. For example, the name of a getter method varies with the field it protects. In the interest of succinctness, it should also be possible to express that the instances of a pattern can exhibit any of several potential characteristics. Likewise, expressing that instances are allowed to exhibit anything but a particular characteristic should also be possible.

CSL4: Provides means for abstraction and reuse among specifications In the interest of reuse, the specification language should support specifications to share groups of common characteristics. While it should be possible to hide the majority of the details of this group from the referring specification, it should also be possible to adapt details to the particular specification in which they are reused. Under these conditions, specifications can be composed in a modular fashion which facilitates the construction of a hierarchical library of patterns.

CSL5: Hides program representation details In the interest of the tool's accessibility, application programmers should not be exposed to the sometimes intricate details of the information carried by its program representation.

This is especially true for behavioral information. As Section 2.5.3 and Sec-

tion 2.5.4 extensively argued for control flow and data flow characteristics respectively, the specification language should shield the user from the intricate details of the behavioral information that supports them. Such information often comes overlaid on an underlying intermediate representation from the domain of optimizing compilers. Context representations add to the complexity of context-sensitive information.

This is also true for non-behavioral information. For instance, the organisation of abstract syntax trees varies among tools. Without proper abstractions, its details must be known to users in order to express syntactic characteristics. With a program representation that combines different kinds of information, the need to hide their intricate details is only aggravated.

2.6.2 Criteria for the Pattern Detection Mechanism

CDM1: Reports elements from the program's source code Given a specification of a pattern's essential characteristics, the detection mechanism of a general-purpose tool is required to report matching elements from the program's source code rather than elements from the internal representation. This criterion not only ensures that users can inspect the tool's results in a straightforward manner, but also facilitates their manipulation by other software engineering tools.

As already argued in Section 2.5.1, establishing a mapping between elements from an intermediate program representation and elements from the program's source code is hard. Interpreting the results from a tool that relies on a canonicalized version of the same code might prove confusing for users as well.

In practice, a specification's matches can be reported in a variety of ways –each attaining this desirable property to a different extent. A straightforward option comprises reporting the line and column numbers elements comprising a match occupy in a program's textual organisation. Modern development environments often provide an application programmer interface through which integrated tools can communicate and share results efficiently in terms of the environment's internal representations.

CDM2: Facilitates user assessment of reported instances To most of the applications enumerated in Section 2.3, the results of a pattern detection tool are only of use once false positives have been filtered out. In case no guarantees about their absence can be given, it is in the interest of the tool's applicability to facilitate user assessment of its results —especially when the sheer amount of instances renders a manual examination of each reported instance infeasible.

The tool should therefore communicate information from which users are able to assess the likelihood that a reported instance is a false positive. Depending on the application domain, users can then concentrate their efforts either on assessing unlikely or on assessing very likely false positives. For instance, a ranking can be devised for all instances the detection mechanism reports.

The likelihood that an individual instance is a false positive can be gauged based on the historical precision of the search strategy used in its detection. All strategies implement a particular trade-off with respect to precision, recall and cost. Section 2.4.3 illustrated this for a pattern comprising two sequentially executed statements.

This likelihood can also vary among the instances identified by a single

search strategy. Conservative approximations relied upon by data flow analyses might lead to the introduction of false positives. The extent to which the strategy is able to assert the presence of a specified data flow characteristic can however be communicated. For instance, must-alias information might be available for local variables within the same method while only may-alias information might be available for others (see Section 2.5.4). This way, the detection mechanism assists users in interpreting behavioral information correctly —just like the specification language hides its intricate details.

CDM3: Supports* implicit *points of variation among pattern instances We already pointed out the need for an expressive specification language in which all of a pattern's characteristics can be expressed together with the *explicit* points of variation among its instances. The semantics of the language should alleviate the otherwise impossible burden of having to enumerate all *implicit* points of variation in a pattern's specification. For instance, a loop can be implemented using a `while` or an `until` control structure.

Realizing the semantics of the specification language, the detection mechanism is able to bridge discrepancies between the evidence necessary to assert a pattern characteristic and the evidence at hand from the program representation. Minor discrepancies are to be expected because it is impossible to account for all implicit variation points in a specification. By interpreting pattern specifications in a non-strict manner, the detection mechanism is able to discover instances that are not implemented exactly as formulated.

However, not all discrepancies originate from implicit variation points. Some discrepancies may originate from possibly imperfect pattern instantiations in the code. Such discrepancies range from differences in the spelling of identifiers over differences in their declared visibility. Nonetheless, the ability to recognize imperfect instances is valuable in many pattern detection applications. For instance, a programmer is not always aware that she is partially implementing a design pattern. As this might come at the cost of additional false positives, this criterion does not require imperfect pattern instances to be identified.

CDM4: Can be extended with user-defined search strategies It is impossible to foresee all problems a general-purpose pattern detection tool might be applied to. Our final criterion has the detection mechanism transparently consider user-defined strategies to complement its predefined ones. This safeguards the separation between the declarative specification of a pattern and the operational search for its instances. Otherwise, users might revert to expressing operational searches tailored to the specifics of a problem.

For the same reason, it is highly desirable to minimize the distance between the tool's pattern specification language and the language in which additional strategies can be implemented. However, it is realistic to relax some of the criteria for the latter. In particular, criterion **CSL1** which requires it to be uniform and criterion **CSL5** which requires all of the intricate details of the program representation to be hidden. Different abstraction levels are appropriate for different customization tasks. However, exposure to raw behavioral information should still be avoided unless explicitly required by the task at hand.

2.6.3 Criteria for the Program Representation

CPR1: Includes behavioral and non-behavioral program information explicitly
Criterion **CSL1** requires behavioral and non-behavioral pattern characteristics to be supported. For each kind of characteristic, Section 2.5 examined which configurations in the design space of a pattern detection tool support it. Realizing the semantics of the specification language, the detection mechanism bridges the gap between the information that is explicitly available in the program representation and the information specifications can refer to. Therefore, as long as the detection mechanism sufficiently compensates its triviality, criterion **CSL1** is already supported by a representation that includes only abstract syntax tree information (see the discourse on its relation to other kinds of program information in Section 2.5.1).

However, under such a configuration, it is nearly impossible to fulfill the requirements of criterion **CDM4**. All search strategies, including the user-defined ones, are tasked with the derivation of all program information they require. By explicitly including this information in the program representation, user-defined search strategies will not have to resort to error-prone derivations of the lacking information. In the interest of accuracy, the representation should therefore explicitly include complete and accurate behavioral and non-behavioral information.

2.7 Conclusion

This chapter initiated our discourse on pattern detection by providing sufficient background to motivate each criterion a general-purpose pattern detection tool should fulfill. Individually these criteria concern one of the key dimensions identified in the design of such a tool: its pattern specification language, its pattern detection mechanism and its representation of the program under investigation. Combined, these criteria ensure that a wide variety of patterns can be detected using descriptive specifications in an expressive language that is accessible to application programmers —regardless of whether these patterns are characterized by syntactic, structural, control flow or data flow characteristics. While emphasising the intricacies of the necessary behavioral program information, the chapter's background sections examined for each kind of characteristic which configurations in the design space of a pattern detection tool support it.

CHAPTER 3

STATE OF THE ART IN PATTERN DETECTION

This chapter presents a survey of the state of the art in tools for detecting user-specified software patterns. The scope of the survey is limited to tools with a declarative specification language and a program representation that carries information obtained through a static analysis. The survey is structured according to the characteristics each tool is primarily tailored to: syntactic, structural, control flow or data flow characteristics. For each characteristic, we complement an in-depth discussion of select tools with a shorter discussion of related tools. For each dimension in the design space of pattern detection tools, a table summarizes the distinctive features of all surveyed tool. Based on this overview, the chapter is concluded with an evaluation of the state of the art on the general-purpose pattern detection criteria identified in the previous chapter.

3.1 Overview of the Surveyed Tools

We initiate our chapter with an overview of the surveyed tools. Table 3.1, Table 3.2, Table 3.3 respectively list the specification language, detection mechanism and program representation of each surveyed tool. For each kind of pattern characteristic, we investigated which combinations support its detection in Section 2.5. Horizontal lines in the overview tables group the tools according to the characteristics they are primarily intended for. Tools listed in **bold** are discussed in-depth.

Table 3.4 evaluates the surveyed tools on the criteria for a general-purpose pattern detection tool introduced in Section 2.6. They are summarized in Table 2.1. The tools marked with a $^\diamond$ are constructed along the lines of the logic meta programming approach to pattern detection (confer Section 4.2). In descending order, entries of the form $+$, $\pm$ and $-$ denote the extent to which each criterion is fulfilled. We will explain the entries in the remainder of this chapter.

```
1 $t $f_decl($*v) {*
2  @*;
3  @{* #{* $f_call(#*) *} *}
4  @*;
5 *}
```

Figure 3.1: A SCRUPLE specification pairing function calls with the function definition they occur in lexically.

3.2 Tools Tailored to Syntactic Characteristics

Syntactic characteristics concern the syntactic structure of code according to the formal grammar of the language it is written in (cf. Section 2.2.1). Section 2.5.1 detailed how pattern detection tools can support the detection of syntactic characteristics.

3.2.1 SCRUPLE: Concrete Syntax Specifications as Code Pattern Automatons

SCRUPLE [Pau92, PP94] is intended as a hypothesis verification and feature localization tool for program understanding and maintenance. It localizes C and PL/AS fragments specified in the concrete syntax of the base program extended with wildcards. Wildcard symbols `$d`, `$t`, `$v`, `$f`, `#` and `@` substitute for an individual declaration, type, variable, function, expression and statement respectively.

The SCRUPLE specification depicted in Figure 3.1 amounts to a naive call-graph extractor for C. It pairs function declarations `$f_decl` with the function applications `$f_call` in its lexical scope. Suffixing a wildcard with an underscore followed by a name transforms it into a meta-variable. By prefixing a wildcard's type encoding symbol with an asterisk, the wildcard will match a collection of program elements (e.g. the parameters `$*v` of the function declaration and the arguments `#*` of the function call). A wildcard suffix of the form `{* *}` allows matches to occur at an arbitrary level of nesting. The function call is allowed to be nested within another expression (`#{* *}`) nested within a statement (`@{* *}`) nested within the function declaration.

Wildcards allow expressing explicit points of variation. Apart from multiple occurrences of the same meta-variable, there are no expressive means to express constraints on explicit variation points such as logic connectives (± for **CSL3**). Rather than a linguistic means to compose specifications, the authors propose a tool solution which allows the matches for one specification to be used as input to another (− for **CSL4**)

Using the concrete syntax of the base program relieves users from having to know the abstract syntax internally relied upon by the tool (+ for **CSL5**). However, specifications lose their resemblance to source code fragments (± for **CSL2**) because of cryptic wildcards. The language does not support expressing behavioral characteristics directly, nor is it expressive enough to express them in terms of syntactic characteristics (− for **CSL1**).

Pattern specifications are compiled into so-called code pattern automatons. These are hierarchical, non-deterministic finite state automatons that consume abstract syntax tree nodes and gather bindings for meta-variables. Each transition is complemented by a navigation function which states how the input to the destina-

tion state can be reached from the node matching the transition condition: move to leftmost child, move to right sibling and move to parent.

While the pattern detection mechanism is not open-ended to start with (− for **CDM4**), its implementation also does not lend itself to possible user amenability as code pattern automatons require users to be aware of how AST nodes relate.

Before serving as input to the code pattern automaton, AST nodes are mapped to a canonical form. At the cost of losing the original code in meta-variable bindings (− for **CDM1**), this supports detecting implicit variation points. The authors illustrate the usefulness of this mapping using the C `do` and `while` statements — even though these are not completely semantically equivalent given the order in which their condition is evaluated with respect to their body. Further support for implicit variation points is limited (± for **CDM3**). The specification in Figure 3.1 has to express explicitly that other statements are allowed before and after the function call. A different specification is necessary to match function declarations with an implicitly declared return type.

SCRUPLE *illustrates that introducing non-native syntax in concrete syntax specifications often detracts from their descriptiveness.* SCRUPLE *also illustrates an early approach to supporting implicit points of variation among a pattern's characteristics. It maps implementation variants to a canonical form. In order to avoid confusing the user, software entities from the original rather than the canonicalized source code should be reported.*

3.2.2 ASTLOG: Prolog with an Implicit Current Object Execution Model

ASTLOG [Cre97] is a Prolog variant designed specifically to localize software idioms through abstract syntax tree traversals. It successfully localizes syntactic bugs in immense C++ programs within the time it takes to have them compiled. It achieves this remarkable performance by foregoing the prevalent transcription of AST nodes into a logic fact base (which separates base program from meta-program look-ups in the database). Node kinds are identified by numeric op-codes for which named aliases are defined (e.g. the op-code alias `#IF`). Aliases are also defined for the integers that identify the children of a node. However, users are exposed to the details of the AST nodes and their reification (− for **CSL5**).

ASTLOG relies on an alternative execution model in which the truth of a predicate is established with respect to an implicit node that is encountered during an abstract syntax tree traversal. Prolog's excellent built-in support for pattern matching, backtracking and query abstraction are nonetheless preserved (+ for **CSL3** and **CSL4**). The following code snippet matches assignment nodes whose left-hand side is a symbol named `"foo"`:

```
1 and(op(#=), kid(#LEFT, asym(sname("foo"))))
```

Upon evaluation, the primitive predicate `op/1` is satisfied whenever its argument is the op-code of the implicit AST node it is evaluated against. The binary predicate `kid(integer-pred, ast-pred)` is satisfied when the current node's child at index `integer-pred` satisfies `ast-pred`.

Detecting software patterns using ASTLOG amounts to launching traversal queries over an AST. Predicates `parent` and `kid` are the primary means to change the current node to a related one. Predicate `op` is the primary means to filter nodes. They also give the resulting parse tree traversals an operational flavour (± for **CSL2**).

```
1 somenode(pred) <- or(pred, kid(_, somenode(pred)));
```

Figure 3.2: ASTLOG definition of a general-purpose tree traversal predicate.

ASTLOG's built-ins can be augmented with user-defined predicates through Prolog's regular predicate definition facilities (+ for **CSL3**. Behavioral characteristics can be expressed through traversals of the AST (± for criterionCSL1). However, users have to enumerate all their implementation variants (− for **CDM3**). The logic rule depicted in Figure 3.2 defines a general-purpose tree traversal predicate. Upon backtracking, the anonymous variable in the rule causes the traversal predicate to be applied recursively to all children of the original node.

ASTLOG *illustrates that logic meta programming tools require an operational traversal of an AST to support syntactic characteristics. To shield the user from traversal details, a higher-order predicate that implements a generic traversal is usually provided.* ASTLOG *goes further by adopting resolution against an implicit current node.*

3.2.3 FUJABA: One Graph Rewrite Rule for Multiple Implementation Variants Weighted by the Expected False Positives

Niere et al. [NWW03, NSW+02] apply a fuzzy [Zad65] variant of the FUJABA [FNT+98] graph rewriting system to detect structural design patterns [GHJV94]. These are specified in terms of UML [Fow97] class diagrams, including relations such as association, delegation and generalization. These structural relations are derived by lower-level rewrite rules over the program's abstract syntax tree. We discuss the tool here as FUJABA itself supports syntactic characteristics, while tools tailored to structural characteristics do not support syntactic characteristics. Graph rewrite rules are sufficiently powerful (+ for **CSL3** and **CSL4**) to express other characteristics in terms of syntactic characteristics (± for **CSL1**).

Figure 3.3 depicts a graph rewrite rule annotating the program's abstract syntax graph with `1N_Delegation` nodes denoting one-to-many delegation pattern instances. Its specification comprises the annotation nodes and edges that are to be added, depicted with the stereotype `<<create>>`, and a sub-graph template identifying locations in the program's graph where they are to be inserted.

Prior to the successive application of rewrite rules, FUJABA's program representation starts out as an abstract syntax tree. From there on, added annotation nodes and edges transform it into a graph. Abstract syntax tree nodes are represented as typed rectangles while annotations are represented as typed ovals. They are connected by named, directional edges. In the lower right corner of Figure 3.3, a `ContainerReference` annotates a class and an attribute of another class. The annotation is connected to the class and the attribute through edges named `field` and `references`.

The rule identifies caller and callee nodes of type `Operation` and verifies whether the caller's class has a container reference to the callee's class. For this, it relies on the `ContainerReference` annotation that is to be discovered by another rewrite rule. It also requires an identifier node of type `PTNodeId` to reside at an arbitrary depth (specified by a `Path` edge) within a loop node of type `PTLoopNode`

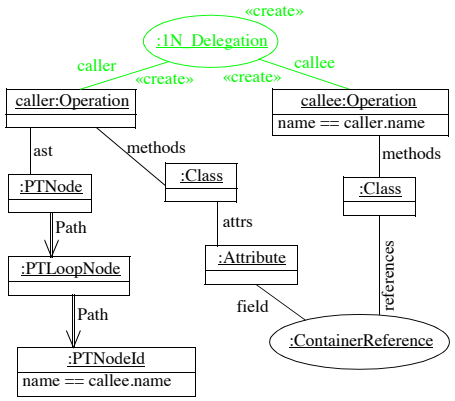


Figure 3.3: An abstract syntax tree consulting FUJABA specification for a one-to-many delegation pattern [NSW+02].

within the caller's method parse tree root reached through an `ast`-edge. An additional node condition requires the identifier, presumed to be part of an invocation node, to match the callee's name which in turn is required to match the caller's name.

The rule annotates exact instances of the one-to-many delegation relation with a precise annotation according to a detailed and verbose grammar rule specification (± for **CSL2**). Another rule is necessary to detect one-to-many delegations implemented through recursion instead of iteration, which increases the overall pattern search space. At the cost of an increase in false positives, a less precise rule that covers both cases could forgo the required loop node or not analyze the caller's body at all.

Instead of specifying a separate graph rewrite rule per implementation variant of such a relation, the authors minimise the search space by specifying a single rule that comprises only the commonalities between the variants (− for **CDM3**). This way, they were able to detect design patterns in Java libraries in the range of 100.000 lines of code. To help the reverse engineer cope with the false positives this deliberate imprecision inadvertently introduces, a weight is added to the resulting rule which represents the anticipated percentage of correct matches.

The fuzzy FUJABA variant [NWW03, NSW+02] supports weighted annotations. The weight of an annotation is computed as the minimum of the weights of all annotations involved in a match and the weight that is associated with the relaxed rule. The associated weight, a number between 0 and 1, is intended to express the reverse engineer's belief in the rule's precision based on personal experience or historical data. Computed annotation weights therefore rank pattern instances according to the expected precision of the rules that were used in their localization (+ for **CDM2**). This allows reverse engineers to filter the false positives caused by the reduction of the total amount of rules. In addition to a weight, each rule is adorned with a threshold above which all of the weights computed for its sub-graph annotations have to lie. This increases the scalability of the approach by not allocating resources on the derivation of pattern instances whose constituents could not be demonstrated with sufficient certainty.

The fuzzy FUJABA variant employs a combination of bottom-up and top-down

rule application strategies in its graph rewriting algorithm. This only affects the sequence in which rules are considered for application. The algorithm starts by scheduling all rules at the lowest level in the hierarchical rule library thus avoiding top-down failures of higher-level rules due to a lack of supporting information. In this bottom-up mode, reverse engineers are allowed to steer the detection process by rejecting or adding intermediate results in an interactive manner. The system switches to a goal-driven, top-down mode as soon as a rule lacks information that could be provided by another rule. This way, the search space of the top-down mode is restricted by the information already derived in bottom-up mode.

When two annotations of the same kind can be derived for the same program sub-graph, only the one with the highest weight is retained. As a rule's weight is an effective upper bound for the weight of all annotations that can be derived from it, rules with a weight lower than the one already associated with a derived annotation therefore need not be considered for application —further restricting the search space.

The fuzzy FUJABA *variant illustrates an alternative to specifying all implementation variants of a structural characteristic: only specifying their commonalities. Focusing on efficiency rather than precision (− for* **CDM3***), extremely relaxed graph rewrite rules serve as pattern specifications which is likely to result in many false positives. However,* FUJABA *illustrates an interesting application of the theory of fuzzy logic: ranking pattern detection results on a theoretical basis (+ for* **CDM2***).*

3.2.4 Other Declarative Tools

Centaur Centaur [BCD+89] represents an early attempt at generating interactive programming environments from a formal language specification. A Prolog-based inference system augments the environment with a type checker, a compiler and an interactive interpreter for debugging purposes. The evaluation of Prolog goals can be controlled from within a Lisp process that runs as a coroutine taking care of most of the graphical user interaction. Relevant to our work is that abstract syntax tree nodes communicated by the Lisp coroutine are transformed into Prolog terms as needed.

Structural search and replace JetBrain's IntelliJ integrated development environment for Java offers an advanced *Structural Search and Replace* [Mos05] feature. Search queries are specified in the concrete syntax of Java that is only extended with untyped meta-variables of the form `$identifier$`. The concrete syntax is matched against an AST, but users are not exposed to its details (+ for **CSL5**). Each template must comprise either an expression, statement sequence or a class definition. The first two template kinds only match exact occurrences in the program's code, while class definition templates also allow non-specified content in their matching base program classes.

Through a graphical user interface, constraints can be applied to meta-variables which severely influence the matching process. A high maximum occurrence constraint on the argument meta-variable in a single-argument method invocation template will for instance also match invocations with multiple arguments. Setting the minimum occurrence constraint of a statement meta-variable in a block template to 0 and its maximum occurrence constraint will have the template match empty blocks as well as blocks whose statements are bound individually to the statement meta-variable.

```
1 (declaration:FUNCTION:$fdef
2   * [expression:FUNCTION:$fcall] *)
```

Figure 3.4: A TAWK expression pairing function calls with the function definition they occur in lexically.

Meta-variables are the only means to express explicit points of variation within a template. Constraints on these points can however be expressed outside of the template (± for **CSL2**). This keeps specifications free from non-native syntax which ensures their descriptiveness (+ for **CSL2**).

PMD Like Checkstyle [Che08], *PMD* [PMD08] examines abstract syntax trees using visitors to detect common bug Java bug patterns and bad coding practices in an operational manner. However, declarative XPath [CD99] expressions can be used to query an XML representation of the program's AST. The following expression will for instance match all `VariableDeclarator` descendants of the root(`//`), whose parent node (`..`) has a type child with name (`@image=`) `Logger`. Users not completely familiar with XPath can build (the sometimes cryptic) path expressions in an exploratory manner using *PMD*'s interactive expression evaluator window which shows matching nodes for an expression.

```
1 //VariableDeclarator[../Type/ReferenceType/ClassOrInterfaceType[@Image='Logger']]
```

AWK derivatives* TAWK *and A*** Inspired by the popular text processing language AWK [AKW88], TAWK [GAM96] programs comprise pattern-action pairs evaluated against a C or MUMPS program's AST. An action consists of C code that is executed when the associated pattern matches. Action code can reference a pattern's meta-variable bindings. Patterns are translated to automatons similar to SCRUPLE's [Pau92, PP94] code pattern automatons. However, TAWK patterns comprise regular abstract syntax tree expressions. Atomic patterns of the form `type:expression:variable` specify a meta-variable to which an AST node of the given type will be bound if one of its child strings matches the given regular expression or literal. Atomic patterns are composed using a syntax reminiscent of regular expressions extended with operators such as $(e\ c_1 \ldots c_n)$ or $[f]$ which respectively match a tree with root matching e and children matching c_i or a tree with a descendant matching f. The TAWK expression depicted in Figure 3.4 can therefore function as pattern in a naive call graph extractor matching function calls at an arbitrary depth within a function declaration.

From the user, TAWK requires detailed knowledge of its internal abstract syntax tree representation such as the tag `FUNCTION` that adorns specific expression and declaration nodes (− for **CSL5**). Its matching semantics is moreover fixed. Pattern abstraction is limited to the definition of a textual macro expanded by the C pre-processor (± for **CSL4**). Representing another tree-based AWK, *A** [LR94] is quite similar to TAWK, but allows custom abstract syntax tree traversals and hence matching semantics to be implemented. These however exhibit a highly non-declarative nature.

LogicAJ2 and GenTL Neither *LogicAJ2* [RKA06] nor the closely related *GenTL* [AK07] were specifically designed with pattern detection applications in mind. However, their respective pointcut and transformation applicability specifications are relevant to our discourse. Both are expressed in logic programming

```
1  pointcut get(?jp,??modifier,?declType,?name)
2     expr(?jp,?name)
3     && ( decl(?field,??modifier ?retType ?name; )
4          || decl(?field,??modifier ?retType ?name = ?v;
5         )
6     && equals(?field, ?jp::ref)
7     && equals(?declType, ?field::parent::type);
```

Figure 3.5: A LogicAJ2 pointcut predicate definition identifying expressions that syntactically reference a field.

languages that feature concrete syntax extensions (+ for **CSL5**) adapting plain Prolog to the respective application domains of aspect oriented programming (AOP [KLM+97]) and program transformation.

The extensions to *LogicAJ2*'s pointcut language comprise three predicates `decl/2`, `stmt/2` and `expr/2`. These select from the candidate joinpoints bound to their first argument those declarations, statements and expressions that match the concrete syntax patterns bound to their second argument. Two types of logic meta-variables can be discerned in concrete syntax patterns. Meta-variables of the form `?var` match one base program element while meta-variables of the form `??var` match an arbitrary amount of elements. This is for instance used to distinguish between patterns intended to match method declarations with a single parameter versus all method declarations. Logic meta-variables can moreover be suffixed by attributes `parent` and `type` which respectively select the parent and type of the value bound to the variable. Attributes are convenient shorthand for logic conditions that would be required to retrieve the information otherwise. The attribute `ref` resolves identifiers and invocation expressions to their static declaration, possibly missing the actual method that is invoked dynamically at run-time. Fine-grained static pointcuts can be defined using the three basic predicates over abstract syntax trees. The pointcut depicted in Figure 3.5 for instance identifies *syntactic* references `?jp` to a field named `?name` by selecting expressions matching the concrete syntax `?name` that resolve to a declaration of the field within the desired type `?declType` [RKA06]. Note how the specification explicitly accounts for field declarations defined with or without an initializing expression (+ for **CSL3**). Unfortunately it misses declarations that define multiple fields of the same type at once. It is unclear from the paper whether field references with an explicit `this` qualifier are identified as well, but *semantic* references through aliases or other layers of indirection are definitely missed as the matching semantics is purely structural (− for **CDM3**).

GenTL [AK07] relies on concrete syntax patterns for expressing where and under which conditions a program transformation applies as well as for specifying any code that is to be generated. It unifies *LogicAJ2*'s concrete syntax predicates in one construct that does not require a pattern's syntactic category to be specified: `?match is [[pattern]]`.

Both tools perform a structural matching on the program's abstract syntax tree and incorporate no behavioral information in the process. Neither paper explicitly defines the semantics used for matching sequences of statements in a concrete syntax pattern. It is therefore not clear whether non-specified statements may appear syntactically between a sequence's constituents. However,

```
equations
[1]    COMPUTE Data-name2 = Data-name1/4
       COMPUTE Data-name3 = Data-name1 - Data-name2*4
       ...
       =
       ...
       DIVIDE Data-name1 BY 4 GIVING
          Data-name2 REMAINDER Data-name3
       END-DIVIDE
```

Figure 3.6: Canonicalizing ASF term rewriting equations from Sellink et al. [SV98]'s native COBOL patterns.

the actual matching semantics is fixed and cannot be altered by the user (− for **CDM4**).

Native patterns Sellink et al. [SV98] apply so-called *native patterns* to the renovation of legacy COBOL programs, more concretely to the identification and correction of instances of the year 2000 problem. These native patterns comprise concrete syntax mixed with wildcards. Interestingly, wildcards correspond to non-terminals in the base language's grammar. Their naming scheme as defined in the language's reference manual is expected to be known by users as wildcards must be named accordingly. List wildcard suffixes + and ∗ provide a limited amount of control over the tool's matching semantics. The tool's pattern specification language can however be generated automatically provided the base language's grammar is expressed in the declarative Syntax Definition Formalism (SDF) [SDF08, VS00, HHKR89]. [1] Its companion Algebraic Specification Language (ASF) [vdBKV07, vdBHKO02, Ber89, DHK96], a term rewriting language, can moreover be used to define semantic equalities between terms matching native patterns. The modularly defined equalities drive problem-specific abstract syntax tree canonicalizations such as the ones depicted in Figure 3.6 which state an equivalence between the COMPUTE and DIVIDE operations involved in leap year calculations [SV98] (± for **CDM3**).

3.2.5 Noteworthy Imperative Tools

Initially designed to check whether a program adheres to predefined coding conventions, *Checkstyle* [Che08] currently also examines the code for common Java bug patterns such as `switch` statements with non-empty cases that fall through. Application-specific checks can be added as abstract syntax tree walkers implementing the Visitor design pattern [GHJV94]. They have no information beyond what's available as an individual compilation unit's AST nodes. *PMD* [PMD08] is similar, but supports in addition declarative XPath [CD99] expressions to query an XML representation of the program's abstract syntax tree. *Spoon* [PNP06] relies on abstract syntax tree traversing visitors not only to implement similar checks, but also to recognize candidates for program transformations of which the code that is to be generated can be specified as plain Java parametrized by meta-variable identifying annotations. *GENOA* [Dev92, Dev99] is a procedural domain-specific language specifically designed to express program query answering traversals over

[1] Section 2.5.1 and Section A.2 discuss it in the context of obtaining abstract syntax trees and structural program representations respectively.

annotated abstract syntax trees. It achieves language and parser independence through an explicit specification of the abstract syntax tree models delivered by an external parser.

3.3 Tools Tailored to Structural Characteristics

Structural characteristics concern the structural organization of the program under investigation (cf. Section 2.2.2). While structural characteristics can be expressed in terms of syntactic characteristics, the tools in this section do not support syntactic characteristics. They employ a coarse-grained program representation that prohibits a complete reconstruction of the program's source code (cf. Section 2.5.2).

3.3.1 PTIDEJ: Deviating Pattern Instances as Solutions to a Relaxed Constraint Satisfaction Problem

PTIDEJ [AACGJ01, GAA01, Gué03] (Pattern Trace Identification, Detection and Enhancement For Java) detects and subsequently corrects implementations that differ slightly from the well-known micro-architectures prescribed by design patterns [GHJV94]. Programs are represented according to an object-oriented meta model whose API forms PTIDEJ's Pattern Description Language (PDL). Its name conveys the fact that the same API is used to describe design patterns as well. The model describing the entities in a design pattern's micro-architecture is called an *abstract model* while models representing the application's architecture are called *concrete models* instead.

The visualisation of the relationship between both models, a mapping of pattern participants to application entities, is the tools' contribution to the design recovery process. The actual mapping is determined by the solution to a constraint satisfaction problem (which has to be specified manually in the Claire [CL96] programming language). Its domain covers the concrete model of the application, while its constraints correspond to the entities in the abstract model of the design pattern and the relationships between them. A solution to the CSP determines the mapping between the entities in the program's concrete model and the design pattern's abstract model.

The Claire extract depicted in Figure 3.7 declares a constraint satisfaction problem whose domain `length(listC)` enumerates the number of classes in the application's concrete model. Its variables `leaves`, `composites`, and `components` capture the *Leaf*, *Composite*, and *Component* entities involved in the Composite design pattern respectively.

Users can specify that the variable assignments of a problem have to adhere to a set of constraints. The built-in constraints govern relationships ranging from inheritance (*StrictInheritanceConstraint* $A < B$, stating that class A must be the superclass of class B), over invocation knowledge (*RelatedClassesConstraint* $A \triangleright B$ stating class A invokes a method of class B) to instance variable types (*PropertyTypeConstraint* $B : A.f = B$ stating that the field f in class A must be of type B).

The Claire extract in Figure 3.7 specifies that the entities assigned to the `Component` variable have to be direct subclasses of the entities assigned to the `Composite` variable. The composite must also be in a composition relation with its leaves. Finally, composites and leaves are required to be different.

```
[problemForCompositePattern() : PalmEnumProblem ->
  let pb := makePalmEnumProblem("Composite Pattern",
                                length(listC),
                                length(listC)),
      leavesTypes := makePtidejIntVar(pb, "LeavesType", ...
      leaves := makePtidejIntVar(pb, "Leave", ...
      composites := makePtidejIntVar(pb, "Composite", ...
      components := makePtidejIntVar(pb, "Component", ...
```

Figure 3.7: Claire extract defining the domain variables involved in a PTIDEJ constraint satisfaction problem for the Composite design pattern.

```
  post(pb,
       makeStrictInheritanceConstraint("Composite,Component |
                                       javaXL.XClass c1,
                                       javaXL.XClass c2 |
                                       c1.setSuperclass(c2.getName());",
                                       composites, components),
       50)
  post(pb, makeCompositionConstraint("throw new RuntimeException(....)",
                                     composites, leaves),
       90)
  post(pb, composites <> leaves, 100)
```

Figure 3.8: Claire extract posting inheritance and composition constraints on the variables involved in a PTIDEJ constraint satisfaction problem for the Composite design pattern.

An explanation-based CSP solver first identifies program entities identical to the micro-architecture put forth by the design pattern the problem encodes. When no solutions can be found, an explanation is given for the solver's failure consisting of unsatisfiable constraints. Individual constraints from this set can subsequently be relaxed. Solutions to the relaxed problem identify source code entities whose relationships only satisfy a subset from the problem constraints. The minimal subset of constraints to which an architecture must adhere is determined either interactively by the user or by weights associated with each constraint.

A weight is associated with each constraint posted to a problem. The strict inheritance constraint is for instance given a weight of 50 in the problem specification depicted in Figure 3.8. These are problem-specific and hence do not establish a hierarchy among the constraints in the constraint library, but rather express a constraint's importance to the problem. A quality metric sorts the problem's distorted solutions based on the weights associated with the relaxed constraints (+ for **CDM3**).

Unsatisfied constraints are considered symptomatic of design defects. In addition to weights, JavaXL [AA01] transformation rules can be associated with a constraint to correct defects of the entities that fail to comply with the constraint. The strict inheritance constraint in Figure 3.8 is for instance given a transformation rule which corrects the inheritance relation.

PTIDEJ *illustrates a potential, but extreme approach to supporting implicit variation points among pattern instances: automatically relaxing characteristics not adhered to by a potential instance. It is better suited to detecting imperfect instantia-*

tions of a pattern, than detecting alternative implementations of a characteristic. PTIDEJ *also illustrates that the ranking computed for a result can be specific to a specification, rather than predetermined by the characteristics in the specification.*

3.3.2 Other Declarative Tools

4Thought The *4Thought* [CMR92] software design tool relies on a graphical Datalog [CGT89] subset called *GraphLog*. Graphs serve as visual representations and specifications of investigated programs and program queries respectively. The actual meta model according to which the program is represented is specific to he problem at hand: a manually extracted entity-relationship model was used to determine package dependencies, while a function call graph was used to partition code into modules.

Pat The *Pat* [KP96, PK98] system extracts coarse-grained structural program information from C++ header files and subsequently queries the resulting logic fact base for structural design patterns using Prolog. The extracted information only pertains to classes, their attributes and operations, and merely a subset of the various ways in which association and aggregation relationships can be declared. As the conditions in Pat's pattern specifications are therefore severely constrained, it suffers from relatively low precision.

Richner's visualisation tool This tool [RD99, Ric02] relies on Prolog to generate user-specified program visualisations. One predicate induces a partitioning of program elements into components visualised as nodes. Another predicate specifies a binary relation that is to be satisfied by any two elements in order for an edge to be drawn between the nodes each individual element is assigned to. In most visualisation queries, the latter predicate queries method invocation traces obtained through a dynamic analysis. The former predicate however usually queries a program's structure represented according to the FAMIX [TDDN00] meta-model.

CodeQuest Designed as a scalable, general-purpose program query tool, *CodeQuest* [HVd06] translates Datalog [CGT89] to SQL queries over a relational database system. The database is populated with information about a Java program's packages, compilation units, types and their members. Binary `hasChild`, `implements` and `extends` relations capture how program elements are related. Information about methods is limited to their signature, the fields they read or write, the types they return and their invocations. *CodeQuest* is tightly integrated with the Eclipse [Ecl08b] integrated development environment, which signals modifications to resources causing the resource's compilation unit and all its children to be re-inserted into the database. *CodeQuest*'s main contribution lies in demonstrating that a modern optimizing database back-end for Datalog allows queries to scale to programs as large as Eclipse. It employs essentially the same program representation as *jQuery*.

jQuery This program navigation plug-in for the Eclipse IDE [De 06, JD03] uses the *TyRuBa* logic programming language to quantify over the base program and configure its user interface. It is a Prolog variant with tabled resolution [RC97, CW96] featuring predicate mode and type annotations. Based on these annotations, *TyRuBa* is able to detect ill-formed queries and order a query's constituents based on heuristics that favour performance.

3.3.3 Noteworthy Imperative Tools

CrocoPat [Bey06, BNL03] supports the detection of structural patterns in a relational representation of a program's structure extracted by a third party, e.g. the inheritance and containment relations among its classes to detect instances of the Composite design pattern [GHJV94]. It combines imperative control flow statements and a transitive closure operation with first-order predicate logic expressions over n-ary relations. Internally it relies on binary decision diagrams (BDD) [Bry92] to represent relations in a compact manner and implement relational operations efficiently. In this it is similar to the on a data flow representation relying PQL [LWL+05a, MLL05, Liv06] which Section 3.5 discusses extensively.

3.4 Tools Tailored to Control Flow Characteristics

Control flow characteristics concern the order in which instructions are executed at run-time (cf. Section problems:characteristics:controlflow). Section 2.5.3 detailed how pattern detection tools can support the detection of control flow characteristics.

3.4.1 Metal: Syntax-Driven Finite State Machine Transitions over Control Flow Graphs

The bug pattern specifications of METAL [ECCH00, CEH02, HCXE02] consist of finite state machines that encode illegal sequences of program instructions. These instructions are specified by concrete syntax patterns (or boolean C expressions) that guard the transitions of the state machine. State machines are simulated along the paths through a control flow graph of which the nodes stem from an AST (± for **CPR1**, + for **CDM1**). They consume the AST nodes encountered on these paths. METAL has been applied successfully to detecting bugs in the Linux kernel [ECCH00], but also to inferring plausible illegal event sequences from existing programs [ECH+01]. It owes a great deal of this flexibility to arbitrary C code that can be associated with the action of a transition.

The specification depicted in Figure 3.9 identifies potential null pointer dereference bugs. To discern meta-variable identifiers from base program identifiers, METAL requires all meta-variables to be declared in advance using the `decl` keyword (e.g. meta-variable `v`). The specification is a template for object-specific state machines with three states `v.unknown`, `v.null` and `v.stop`. Each object-specific state machine tracks the state of a pointer `v`. The specification itself is, in contrast, a global state machine. It has one creation state `start` on line 6. When its transition guard is applicable, a new object-specific state machine is instantiated to track the pointer `v`. There is a transition from the global `start` state to the object-specific `v.unknown` state. The transition is applicable if its transition guard, the pattern `v = kmalloc(x,y)`, matches the current AST node. The object-specific state machine starts in the `v.unknown` state because `v` is allocated using an instruction that offers no null-ness guarantees. Whenever a use of the pointer is encountered in the `v.unknown` state or in the `v.null` state, the object-specific machine's execution is stopped and an error is reported. The instructions that constitute a use are enumerated on line 4 as a disjunction of concrete syntax patterns.

The `v.unknown` state on line 7 features path-specific transitions. Their destination states vary according to the branch that is taken out of the conditional node

```
1  sm null_checker local {
2     state decl any_pointer v;
3     decl any_expr x,y;
4     pat use = { *(any *)v } || { memset(v,x,y) }
5               || { v + x } || { v - x };
6     start: { v = kmalloc(x,y) } ==> v.unknown;
7     v.unknown:   { (v == NULL) } ==> true=v.null, false=v.stop
8                | { (v != NULL) } ==> true=v.stop, false=v.null;
9     v.null, v.unknown: use ==> v.stop,
10     { v_err("NULL", v, "Using \"$name\" illegaly!"); }
11    ;
12 }
```

Figure 3.9: A METAL finite state machine specification identifying possible null pointer dereferences.

identified by their pattern guards. The state of `v` is definitely null on the `true` path out of a positive comparison against null (line 7), while it no longer needs to be tracked on the `false` path —and vice versa for negative comparisons (line 8).

Transition guards consist of conjunctions and disjunctions of concrete syntax (+ for **CSL5**) extended with meta-variables. They can be complemented by callouts to C. The following transition guard only triggers when a call to a debug function is encountered:

```
1  { call(-1,v) } && ${is_debug_call(call)}
```

Metal's support for checker composition amounts to the sequential execution of the constituent state machines where each individual machine can annotate the program's abstract syntax tree with data that is to be shared (± for **CSL4**).

The state machine simulator implements a depth-first control flow graph traversal, backtracking to the latest branch point whenever a path has been exhausted. Each encountered abstract syntax tree node is matched against the guards of the current state's transitions, rendering the algorithm flow-sensitive. Function calls are followed from multiple call sites and only returned from when all paths through the callee have been exhausted. The resulting analysis is therefore interprocedural and context-sensitive. However, the `local` qualifier adorning the null checker example in Figure 3.9 keeps the analysis intraprocedural.

Multiple occurrences of the same meta-variable in a specification must stand for equivalent abstract syntax trees. Arbitrary expression aliasing is ignored, but the state machine simulator keeps track of variable synonyms along a path when it encounters one variable being assigned to another. This provides limited support for detecting implicit variation points (± for **CDM3**).

State caches at join points in the flow graph eliminate redundant paths along which machines reach a join point in the same state. They also take care of termination in the presence of back edges. By keeping track of which branch was followed out of a conditional expression, infeasible paths can be pruned whenever a branch out of another conditional is excluded by the knowledge gathered along the path.

Based on the effort a manual inspection would require, METAL classifies local bug pattern instances above interprocedural ones (+ for **CDM2**). In a similar vein, instances without synonyms are considered a higher priority than the ones without. Within these severity classes, instances are sorted according to the amount of

```
1 from "close(%F)" to "read(%F,%_ )" avoid "%F=open(%_ ,%_ )"

2 from "%X=malloc(%_)" to "*%X" avoid +"%X!=0" or -"%X==0"

3 from "%T %X" | TYPE_P(T) && TREE_CODE(X)==VAR_DECL &&
4                DECL_SIZE(X)>1024
```

Figure 3.10: CONDATE constrained reachability queries identifying reads from a closed file, potential null pointer dereferences and large variable declarations respectively.

lines they span and the number of conditionals encountered along the erroneous path. A statistical ranking augments this generic ranking by preferring checks that are more often adhered to than violated.

METAL *has proven successful and influential in bug pattern detection.* METAL *also illustrates a multifaceted ranking of results according to the effort their manual inspection would require.*

3.4.2 Condate: Constrained Control Flow Reachability Queries between Unparsed Patterns

CONDATE [Vol06a, Vol06b] represents a compiler-integrated approach to the permanent detection of user-defined bug patterns. Bug patterns are specified as a sequence of program events which are identified by concrete syntax patterns with meta-variables. To stimulate continuous checking at compile-time, sequencing properties are restricted to a class that is checkable both in linear time and space. While deliberately less powerful than the heavyweight stand-alone checkers it intends to complement, such a compiler-integrated approach automatically offers continuous checking and the possibility to reuse program analyses already present in industrial-strength compilers.

A bug pattern has to be specified as a constrained reachability query (CRQ) of the form *"Is there a path from a program fragment f to a program fragment t avoiding: fragments v, successful tests v_t and unsuccessful tests v_e?"* where $\langle f, t, v, v_t, v_e \rangle$ are respectively the $\langle$*from, to, avoid, avoid-then, avoid-else*$\rangle$ patterns of which all but the *from* pattern can be omitted. These patterns match atomic expressions or statements in the program's control flow graph. They are specified as quoted strings containing concrete syntax with meta-variables preceded by the escape character `%` or the anonymous meta-variable `%_` (+ for **CSL2**).

The first CRQ depicted in Figure 3.10 captures the pattern describing a read from a closed file in C which manifests itself when there is a path from an expression closing a file `%F` to an expression reading from this same file without it being reopened along this path. Note that the meta-variable `%F` intends to relate the different syntactic patterns along the control flow path to the same file. All occurrences of a meta-variable must stand for structurally equivalent subfragments (– for **CDM3**). The semantics of the programming language is not taken into account.

By expressing whether the positive (`then`) or negative (`else`) branch out of a decision node in the control flow graph is to be avoided, the *avoid-then* and *avoid-else* syntactic patterns allow for more fine-grained control over the edges constituting the paths matching a CRQ. This is for instance important in the second CRQ de-

picted in Figure 3.10 that matches potential null pointer dereferences. Only dereferences `*%X` encountered on a path where the boolean expression `%X!=0` evaluates to false constitute an error.

Finally, a CRQ's matches can be further constrained using boolean expressions that refer to the compiler's internals — provided users are familiar with its meta model. As meta-variables are untyped, the *from* pattern in the third CRQ depicted in Figure 3.10 would match every occurrence of two program constructs, were it not for the boolean expression restricting the bindings to large variable declarations.

The specification language provides no mechanism to reuse individual CRQs (– for **CSL4**). Although syntactic patterns within a CRQ can comprise disjunctions, this correlating the choices among different patterns is not possible (± **CSL3**). The following CRQ does check whether all locks are eventually released, but it cannot check whether each lock is released using the function $unlock_i$ corresponding to the one that acquired the lock $lock_i$:

```
1 from  "lock_1(%X)" or ... "lock_N(%X)" to  "return"
2 avoid "unlock_1(%X)" or ... "unlock_n(%X)"
```

A CRQ $\langle f, t, v, v_t, v_e \rangle$ corresponds to the regular path expression $f[\hat{}v+v_t-v_e]^* y$ which matches any edge leaving a node matching the syntactic pattern f, followed by an arbitrary concatenation of edges —that individually do not match any edge leaving a node matching v, nor a positive (`then`) edge leaving a decision node matching v_t nor a negative (`else`) edge leaving a decision node matching v_e— which leads to a node matching the syntactic pattern y.

Syntactic patterns may contain meta-variables. Such expressions are existentially qualified parametric regular path expressions. An algorithm for evaluating such expressions with respect to an intraprocedural control flow graph has been studied in detail in [LRY+04]. CONDATE'S CRQs are a particular class of regular path expressions for which the algorithm requires only linear running time and space. The generated automaton that is executed along the paths in the control flow graph is of the same size for all CRQs. CONDATE furthermore imposes the restriction that all meta-variables must be instantiated in the positive *from* pattern f, eliminating the costly possibility of having different substitutions match the same regular path expression to the same path.

The nodes in CONDATE'S control flow graphs do not constitute nodes from an AST (– for **CDM1**), but rather nodes from the GCC compiler's GIMPLE [Mer03] intermediate representation. This representation introduces temporary variables to ensure all instructions involve maximum three addresses. Their definition is conceptually inlined in an attempt to reconstruct original syntax of the expression when matching a concrete syntax pattern against a GIMPLE node. This is not always successful (– for **CSL1**).[2]

CONDATE's matching algorithm for syntactic patterns only requires an *unparser for the base programming* language (pretty-printer) rather than a parser for the pattern language (i.e. unparsed patterns [VR08]) The technique recursively unparses the base program's abstract syntax tree (AST) to compare it against the pattern which is kept as a string. In order to ensure that meta-variables are bound to AST subtrees rather than plain strings, unparsing is performed lazily, one tree level at a time keeping subtrees intact. However, meta-parentheses `%(` and `%)` might have

[2] A syntactically different, yet semantically equivalent expression might be reconstructed from the components of the original expression.

to be introduced in a pattern to force unparse steps. For instance, to take the subtraction operator's left associativity into account: `%w = %(%(%x - %y%) - %z%)`. Awareness of the compiler's internal AST structure is the price users have to pay for the implementer's convenience (± for **CSL5** in spite of the use of concrete syntax for specifications).

Condate *illustrates that many bug patterns can be specified as descriptive, existential reachability queries.* Condate *also illustrates that the benefits of concrete syntax specifications are easily negated when too many non-native constructs are introduced. Furthermore, it illustrates that many intermediate program representations do not suffice to adequately support syntactic characteristics.*

3.4.3 Other Declarative Tools

Parametric regular path expressions Liu et al. [LRY+04] present a detailed study of an algorithm for evaluating parametric regular path expression with respect to an intraprocedural control flow graph. These are regular expressions, parametrized by meta-variables, over the labels encountered along one (existentially quantified) or all (universally quantified) paths in the control flow graph. With analysis-specific abstractions of program instructions such as **def**(x) substituting for the usual control flow graph labels, it has also been successfully applied to classical problems such as the search for uninitialized variables: [^**def**(x)]***use**(x). The algorithm translates the regular path expression to an automaton which is executed along paths in the control flow graph, following a transition whenever its label matches the current graph's edge. It computes all tuples $\langle v, s, \theta \rangle$ consisting of a reachable node v, reachable automaton state s and corresponding meta-variable substitution θ that are reachable from a given starting node and state. *Condate*'s (Section 3.4.2) CRQs comprise a restricted class of regular path expressions for which the algorithm requires only linear running time and space.

MJ *MJ* [BE03] is an implementation of *Metal*'s (Section 3.4.1) finite-state machine based bug pattern specification language and its corresponding static checker for Java. In contrast to the original implementation for C, an *MJ* automaton does not consume abstract syntax tree nodes in their control flow graph derived execution order. It rather consumes nodes from an intermediate register-based bytecode representation computed by the Joeq [Wha03] virtual machine and compiler framework (− for **CDM1**). Identifying concrete syntax patterns, which guard each automaton's transitions, in this intermediate representation is therefore much more involved. *MJ*'s mapping can rely on optional debug information embedded in class files such as line numbers and a method's local variable names. The drawbacks and advantages of using a mapping between concrete source code and an intermediate representation have already been discussed for *Condate* (− for **CSL1**).

TRANS, *Path Logic Programming and JunGL* As powerful program transformation specification languages, neither TRANS [LdM01, Lac03], *Path Logic Programming* [DdMS02], nor *JunGL* [VEdM06] is intended for general-purpose software pattern detection. Their specifications must however stipulate precisely under which conditions a transformation is applicable and most of these conditions take the form of control flow graph constraints.

TRANS [LdM01, Lac03] is based on conditional rewrite rules of which the

left-hand side comprises sequences of statements in the base program's concrete syntax extended with meta-variables. The concrete syntax patterns are matched against basic blocks in the program's control flow graph to identify a transformation's candidate subjects. These are further refined by additional reachability constraints over the graph expressed as *computational tree logic* (CTL) formulae [CES86], verified by a model checker, that are associated with each rewrite rule. In the case of eliminating unused variables by a transformation that removes the defining node, these are for instance used to stipulate that for all of the node's successors there is no path on which the variable x is eventually used: $AX(\neg E(\texttt{True}U\texttt{use}(x)))$. Such temporal path expressions can be used as precise specifications of control flow characteristics, but are hard to get right and read ($\pm$ for **CSL2**). This view is shared by Dwyer et al. [DAC99] who propose a higher-level pattern language instead to describe how program events are related. Moroever, in [dLW03] de Moor et al. motivate their exploration of regular path expressions by the observation that most of their TRANS examples don't exploit the full power of temporal logic.

Existential and universal regular path expression primitive predicates `exists P (A,B)` and `all P (A,B)` are the logic programming extensions proposed by path logic programming [DdMS02]. They are compiled to Prolog with tabled resolution according to the algorithm in [dLW03]. A logic term representing the transformation action to undertake is associated with successful queries. The regular path primitive predicates are satisfied when the sequence of labels encountered on one path (and respectively all paths) between the program's control flow graph nodes A and B adheres to the language defined by the regular path expression P. Its alphabet consists of curly braces housing a conjunction of logic predicates an individual edge label should adhere to.

```
1 unused_from(X, N) :-
2   all ({ ‘unused_other_than_at(N,X) }*;
3        (ϵ + { ‘def(X) }; { }*))
4       (N, exit).
```

The above path logic programming query from [DdMS02] checks whether `X` is an unused variable defined by a given node `N` if all paths from `N` to the `exit` node are either completely described by a potentially empty sequence of edges pointing to nodes that don't use `X` (with the exception of `N` itself) or are described by such a sequence followed by a redefinition of the variable. Note that `{}` matches any edge label while ϵ matches the empty path. The ticked (`‘`) logic predicates take the current edge label against which they are evaluated as an additional implicit argument.

While the above approaches work on an intermediate program representation ($-$ for **CSL1** and **CDM1**), *JunGL* [VEdM06] works on abstract syntax trees as it is intended as a specification language for refactorings rather than optimizing transformations. It extends the ML programming language with stream comprehensions `{?x | Q}` ranging over all bindings for the meta-variable `?x` that satisfy the Datalog query `Q`. *JunGL* supports regular path expressions to express the preconditions of a refactoring and to add derived edges to the control flow graph in a demand-driven way. The following *JunGL* extract from [VEdM06] is, for instance, used to resolve `SimpleName` nodes to a method parameter declaration. Here, square brackets distinguish predicates over control flow graph nodes from predicates over edge labels. The declaration is thus reached by following one or more parent edges to a method declaration node

`?m` and taking the child edge that leads to a parameter declaration `?dec` with the correct name. The omitted query disjuncts (...) handle the case where the `SimpleName` refers to a local variable declaration or to a field declaration analogously.

```
1 let matches = { ?dec | ...
2     ([from] parent+ [?m:Kind("MethodDecl")]
3      child [?dec:Kind("ParamDecl")] &
4     ?dec.name == name) ... }
```

3.4.4 Noteworthy Imperative Tools

FindBugs [HP04] is effective at localizing potential bug pattern instances in Java programs. Application-specific checks can be added by extending a Visitor design pattern [GHJV94] implementation. In this it is similar to both PMD [PMD08] and Checkstyle [Che08]. However, rather than walking an abstract syntax tree, FindBugs' visitors traverse class files. Bytecode instructions can either be scanned linearly without regard for control flow or along a control flow graph. Some checks, such as the check for potential null pointer dereferences, rely on a simple intraprocedural data flow analysis. However, defining one's own checker is hard as this requires expert knowledge of Java bytecode instructions. Much of the search strategy and related bookkeeping has to be implemented imperatively.

3.5 Tools Tailored to Data Flow Characteristics

Data flow characteristics concern the range of possible run-time values expressions can assume, as well as their origin and how they are related (cf. Section 2.2.4). Section 2.5.4 detailed how pattern detection tools can support the detection of data flow characteristics.

3.5.1 JTL: Java-like Datalog Syntax over Structural and Intraprocedural Data Flow Information

The Datalog [CGT89] derivative JTL (Java Tools Language) [CGM06b] has been used primarily to detect [CGM06a] what the authors call μ-patterns [GM05]. This requires support for structural characteristics (cf. Section 2.1). In addition, JTL supports data flow characteristics to enable detecting unread parameters, unused locals, etc ...

Due to deliberate departures from the regular Datalog syntax, many JTL specifications have a query-by-example flavour (+ for **CSL2**). The solutions to the following query, for instance, consist of all parameter-less public abstract methods that are declared to return `void`:

```
1 public abstract void ()
```

JTL predicates have an implicit subject variable and are named after the Java keywords that qualify matching subjects. Juxtaposition amounts to a conjunction in which the subject variable is shared by the conjuncts. Note that the argument list in the query is actually a predicate succeeding for empty argument lists. Also note that the name of the method is missing from the query. Although the query has a query-by-example flavour, its syntax is far from the concrete syntax of Java.

```
1 service := public !static method;
2 statemachine := interface offers: {
3   service => ();
4   exists service;
5 };
```

Figure 3.11: JTL specification for the *state machine* μ-pattern [GM05].

JTL specifications for more complex patterns often rely on set quantification abstractions through which the underlying operational semantics become apparent. The unary `statemachine` predicate defined in Figure 3.11 captures the specification of the state machine μ-pattern [GM05]. It is defined as an interface that only offers parameter-less instance methods (i.e. a purely structural characterization). Set quantifications consist of a generator (e.g. `offers:`) which generates a set of program entities against which quantified queries are evaluated (lines 3 and 4). The definition relies on the `=>` quantifier which is satisfied whenever every set element that adheres to its left hand side also adheres to the query on its right hand side. It is used most often in combination with the `exists` quantifier to make sure there is at least one element that satisfies its left hand side. The query-by-example flavour of the query `class { int field; }` is due to default values for quantifiers and generators. It is equivalent to the query `class members: { exists int  field; }`.

In contrast to the purely structural queries presented so far, the `data_manager` predicate defined in Figure 3.12 relies on intra-procedural data flow information. It detects classes whose instance methods consist solely of getter and setter methods (i.e. the data manager μ-pattern [GM05]). This is specified at lines 2–4. The `getter` predicate is satisfied by non-`void`, parameter-less methods whose `return` operands all originate from a bytecode retrieval of a field `F` within the method's class (line 6). The binding for `S` (called a scratch) is an anonymous representation of the operand's run-time value (i.e. it carries no information about the concrete value), but can be passed to other predicates to express the transitive data flow dependency between the field and the `return` operand ($\pm$ for **CSL5**). Note that dependencies established by field aliasing or inter-procedural flow are missed. The `setter` predicate, on the other hand, ensures that all of a method's bytecode field assignments are to the same field and have an intra-procedural data flow dependency on one of the method's parameters (lien 9).

Data flow characteristics are supported through predicates that range over so-called scratches. Values originating from the method's parameters, receiver, operand stack or its local variables are represented by the unary predicates `parameter`, `this`, `temp` and `local` respectively. The binary predicates `from` and `func` connect these values together by stating that one is obtained from the other through an assignment and an arithmetical computation respectively. Predicates such as `getfield` and `get_invokespecial` state that a value is obtained through a field reference or a method invocation. Branches in a method's control flow can generate multiple occurrences of the same value in these relations.

Structural characteristics are supported through binary predicates such as `extends` and `members`. Control flow characteristics are not supported. Method bodies are represented using binary predicates such as `invokes_virtual` and `invokes_special` (i.e. sets of bytecode instructions, $\pm$ for **CSL5**). Syntactic in-

```
1  data_manager := class is C offers: {
2    exist instance field;
3    exist service;
4    service => [setter | getter];

5    getter := !void () returned: {
6      all from* S, S getfield F, C holds F;
7    };

8    setter := void (_), C offers F, F field {
9      putfield _ => putfield F, from* P, P parameter;
10     exists putfield;
11   };
12 };
```

Figure 3.12: Definitions for the data flow incorporating JTL predicates that identify instances of the data manager μ-pattern.

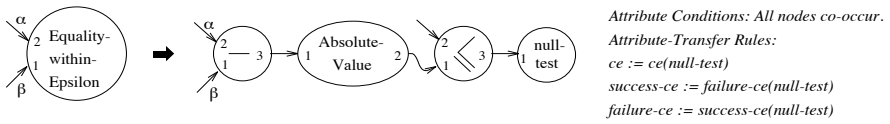


Figure 3.13: GRASPR flow graph grammar rule encoding the equality-within-ϵ idiom [Wil94].

formation is not supported either (– for **CSL1**).

JTL queries are translated to Datalog [CGT89] in a syntax-directed way. This entails handling every predicate's implicit subject variable and translating disjunction to auxiliary rules that implement the same predicate. As soon as a generator's set has been enumerated and each quantifier's query subject is set to the elements of this set, evaluation of quantifiers is straightforward. Evaluation of the existential quantifier is trivial as Datalog queries are already existentially qualified. Negation is used to implement the universal quantifier in terms of the existential one. The translation of JTL to Datalog is fixed. Changing a rather complex Java implementation comprises the end-users' only means to amend these semantics (– for **CDM4**).

JTL *illustrates that a carefully chosen surface syntax that resembles the syntax of the base program can make logic specifications more familiar to developers. However, the specifications for all but the simplest patterns lack any real resemblance to source code excerpts.*

3.5.2 GRASPR: Idioms as Attributed Data Flow Sub-graph Isomorphisms

GRASPR (Graph-based System for Program Recognition) [RW90, Wil92, Wil93, Wil94] is an early program recognition tool for Lisp. Given a hierarchical library of software patterns describing algorithms and data structures, it produces a forest of design trees that specify the implementation relationships among any recognized higher and lower-level software patterns.

It uses a very simple directed, acyclic graph as program representation. Its nodes represent the program's primitive operations. Input and output ports are

associated with nodes. Data flow edges between instructions are represented as edges between ports of the corresponding nodes. Edges fanning out of a node indicate that there are multiple consumers for the result of its operation. Sink nodes in the graph (i.e. nodes without outgoing edges) represent conditional tests.

GRASPR's design trees are in fact parse trees obtained by parsing the program's data flow graph using a context-free graph grammar encoding of the pattern library. Figure 3.13 depicts GRASPR's production rule for the low-level *equality-within-ϵ* idiom which tests whether two values are less than ϵ apart. It specifies how its left-hand side node can be replaced by the data flow subgraph on its right-hand side. The atomic instructions on the right hand side are `-`, `<=` and `null-test`. The latter is a sink node (i.e. no outgoing edges). Ports are depicted as numbers on the corresponding nodes. Ports on either side of the rule are mapped according to an embedding relation. Labels α and β graphically depict the rule's embedding relation.

The ability to refer to non-terminals on a rule's right-hand side provides a means of abstraction for alternative pattern implementations (+ for **CSL4**). The *equality-within-ϵ* rule's right-hand side refers to the *absolute-value* idiom. Production rules are hence organized in a hierarchical library which allows implementation relationships among patterns to be captured in derivation trees.

Attributes on graph nodes are used to encode control flow information in the form of named control environments. Each control environment refers either to a failure or success environment of a conditional test (sink nodes in the graph). Left-hand side node attributes are transferred from the nodes matching the rule's right-hand side. The actual transfer is determined by attribute transfer rules. As the *equality-within-ϵ* pattern is evaluated under the same control flow conditions as its inequality check, they share the same control environment attribute (`ce:=ce(null-test)`). Being a boolean predicate and hence a sink node itself, it is also annotated with a success and failure control environment. These are the reverse of its own sink node. Attribute conditions further constrain potential matches for a rule's right-hand side.

GRASPR's flow graph canonizes implementation variants of the same data flow characteristics (+ for **CDM3**). The order of operations in this representation is determined by their data flow dependency. Programs that differ only in the ordering of independent operations are mapped to the same representation. This is for instance the case for the snippets below, adapted from the original in [Our89]. This precludes GRASPR from being used in API conformance checking, but overcomes non-essential control flow variations in the recognition of low-level software idioms.

```
1 (defun FGH (x)          (defun FHG (x)
2   (let ((z (F x)))          (H (F x))
3     (G x)                   (G x)
4     (H z)                   x)
5     x))
```

The program's flow graph is parsed in accordance with the pattern library's attributed graph grammar. To recognize low-level patterns that are not part of a higher design, parsing is performed in a bottom-up manner —treating all non-terminal nodes as possible start nodes and ignoring any unrecognizable flow surrounding candidate idioms. The detection process therefore amounts to computing subgraph isomorphisms. The parser maintains a chart of both partially and completely recognized items which is extended by continuously combining items.

Chart monitors can intervene in the process. For instance, to further canonize the flow graph on-the-fly (+ for **CDM3**) or consult the user for advice.

GRASPR's approach to idiom detection requires a graph isomorphism to be established between a pattern and an instance in the program's flow graph. A precise flow graph of both is vital to the approach. Prior to recognition, the program under investigation has to be converted manually to a restricted functional programming style in order to circumvent the prototype's inability to handle side-effecting operations, tree recursions and function arguments. Pattern specification involves a manual data flow analysis and its subsequent transcription to a graph grammar production rule. This is especially hard when arbitrary, user-defined functions can interrupt the flow between the idiom's primitive operations.

GRASPR *is of historical relevance as it is one of the earliest pattern detection tools that uses a data flow representation. It illustrates how data flow information supports implicit points of variation among instance. Interestingly, the order of operations in this representation is completely induced by their data flow dependency rather than the one that is specified in the program's code. At the same time,* GRASPR *illustrates the need for a specification language in which not all of a pattern's characteristics need to be specified in excruciating detail.*

3.5.3 PQL: Concrete Syntax Resembling Data Flow Queries

PQL (Program Query Language) [LWL+05a, MLL05, Liv06] supports the specification of control flow characteristics and data flow characteristics in a unified language. However, PQL detects the former using dynamic analysis and the latter using static analysis (cf. Section 2.4.3). Structural and syntactic characteristics are not supported (− for **CSL1**).

The PQL prototype has successfully identified errors related to security and object persistence in web applications as well as instances of the lapsed listener problem (cf. Section 2.1) in real-life Java applications.

Ignoring the actual ordering between run-time instructions, the static analysis identifies the *potential* matches for a query. It relies on a context-sensitive points-to analysis [WL04] to approximate the run-time objects operated on by the instructions (cf. Section 2.5.4). The subsequent dynamic analysis only needs to verify the ordering of the actual instructions and the aliasing relations between their associated objects. As program points deemed irrelevant by the static analysis must not be instrumented, the run-time overhead of the dynamic analysis can be kept to a minimum. Whenever a match does occur at run-time, user-provided actions can correct the error on-the-fly.

The scope of this dissertation is restricted to statically obtained program representations. We will only discuss the static resolution of *PQL* queries. It ignores the execution order of instructions. It solely relies on the information that is vital to reasoning about object-oriented programs: points-to information and the invocation graphs that can be derived from it (cf. Section 2.5.4).

The program is represented as a set of Datalog [CGT89] relations. Operations operating on primitive data types are ignored. Input relations are populated from the program's bytecode, while a set of Datalog rules derive relations that complement the input relations with approximations of the heap objects their object attributes may point to and the method declarations an invocation may resolve to at run-time. These derived relations are parametrized by a static approximation of the callers on the program's run-time call stack and are hence context-sensitive.

```
1 simpleSQLInjection(b1, b2, h) :-
2   IE(c1, b1, _, "getParameter"),
3   ret(b1, v1),
4   vP(c1, v1, h),
5   IE(c2, b2, _, "execute"),
6   actual(b2, 1, v2),
7   vP(c2, v2, h).
```

Figure 3.14: Datalog rule that identifies straightforward SQL injections using the predicates from PQL's program representation.

The following table lists the relations related to a method invocation at bytecode b. Additional relations capture assignments, field and array loads, field and array stores and direct object allocations.

$actual(b, z, v)$	variable v is the invocation's zth actual argument
$ret(b, v)$	variable v stores the result returned by the invocation
$IE(c_1, b, c_2, m)$	in context c_1, method m may be invoked in context c_2
$vP(c, v, h)$	in conctext c, variable v may point to heap object h

Many patterns that are primarily characterized by data flow characteristics can already be expressed as direct Datalog rules over the above relations. Figure 3.14 defines a Datalog rule that identifies straightforward SQL injections (cf. Section 2.1) where a heap object h is the result of an invocation of a method named `getParameter` and is passed as the first argument to an invocation of a method named `execute`:

Note that this static rule does not enforce any ordering between the two calls. Their presence in the bytecode is sufficient. Rather than requiring the return variable `v1` and actual parameter variable `v2` to be the same, the rule only requires them to possibly point to the same heap object `h`. This abstracts away from syntactically differing implementation variants through which this aliasing relation can be established. However, the user has to manage invocation contexts `c2` and `c1` manually and work directly on the program representation.

PQL queries therefore specify a bug pattern as a set of program events connected by sequencing operators. Again, these operators are only taken into account by the dynamic analysis. The events themselves are specified in a Java-like syntax (+ for **CSL5**) with meta-variables (+ for **CSL3**). The PQL query depicted in Figure 3.15 represents a more complete SQL injection specification which considers strings derived from the `getParameter` source tainted as well:

The object meta-variable in a query have to be declared as either local or output variables (qualified by `uses` and `returns` respectively) or as argument variables passed in from other queries (line 9). Each object meta-variable is typed and will match only heap object approximations that can be cast to that type. A `!` prefix will take the complement of such a type declaration. Every occurrence of the same object meta-variable must match the same heap object approximation. Member meta-variables will match methods and fields rather than objects and must be declared with the pattern their name has to adhere to.

Queries can be named and called from other queries. An extra-native operator `:=` has to be used to bind the returned value to a meta-variable. The `main` query in Figure 3.15 relies on another `derivedString` query to determine whether the argument to the `execute` invocation is derived from user input. This particular query is recursive: its base case returns its argument while the recursive cases

```
1  query main()
2      returns object Object source, tainted;
3      uses object java.sql.Statement stmt;
4  matches {
5      source = req.getParameter();
6      tainted := derivedString(source);
7      stmt.execute(tainted);
8  }
9  query derivedString(object Object x)
10     returns object Object y;
11     uses object Object temp;
12 matches
13     y := x
14     | { temp.append(x); y := derivedString(temp); }
15     | { temp = x.toString(); y := derivedString(temp); }
```

Figure 3.15: A more complete specification of the SQL injection bug pattern in PQL's specialized syntax.

consider transitive alternations of `append` and `toString` invocations. Recursion is used in PQL to either define such recursive event patterns or, when field member meta-variables are involved, recursive object relations.

There is a straightforward mapping of PQL queries to Datalog rules. Primitive PQL statements over heap objects are mapped to Datalog predicates over bytecodes and program variables —augmented with a lookup of program variables in the points-to relation `vP`. The `:=` operator maps to an equality test of heap object approximations. Individual operands of a disjunction map to separate Datalog rules implementing the same predicate. Any control flow implied by statements joined by the sequencing operator is ignored as the latter simply maps to a Datalog conjunction. Only the `within` control flow construct is supported. It requires matching bytecodes to occur in methods that are called transitively from its operand.

The actual PQL semantics are fixed. Knowledgeable end-users could in theory alter this mapping by modifying the Java visitors that generate the Datalog rules (– for **CDM4**).

The static resolution of PQL queries relies heavily on a precise approximation of the heap objects that are involved in every possible program execution. A context-sensitive points-to analysis [WL04], implemented by a handful of Datalog rules, provides the necessary precision. The analysis is cloning-based: per call site it creates a unique method clone to which a regular context-insensitive analysis is applied resulting in a points-to relation that is parametrized by explicitly represented calling contexts.

The entailed explosion of context information is managed only by a particularly clever Datalog implementation which exploits similarities between contexts. This implementation, bddbddb (Binary Decision Diagram Based Deductive Database) [WACL05], encodes Datalog relations as boolean functions over tuples of domain values enumerated as binary numbers. These can be represented efficiently using binary decision diagrams (BDD) [Bry92]. Resolution of Datalog queries with a finite domain and stratified negation maps easily to a sequence of BDD operations . Operating on an entire relation at the same time, their cost depends on the shape of the BDD graphs encoding the relation rather than the relation's population size.

```
1 method("ResultSet Statement.executeQuery(String)", ExecMeth),
2 call(ExecMeth, Receiver, ExecCall, QueryStringLocal),
3 assignment(Target, ExecCall, _),
4 member(QSLocal, QueryStringLocal),
5 local_constant_def(QSLocal, QueryString),
6 findall(Column, columnUsed(Target, Column), Columns)
```

Figure 3.16: DeepWeaver pointcut conditions identifying expensive database queries.

Computation of the points-to information and the dependent PQL queries is performed exhaustively in a bottom-up manner.

PQL *illustrates that it is possible to support precise data flow characteristics without exposing users to the details of its enabling analysis. The specification language resembles a subset of Java's concrete syntax: those instructions that influence data flow characteristics directly.*

3.5.4 Other Declarative Tools

DeepWeaver As a bytecode transformation and program optimization tool, *DeepWeaver* [FKI+07] is not intended for software pattern detection per se. Its transformations are however expressed in the aspect oriented programming paradigm (AOP) [KLM+97] and therefore include a *pointcut* part which identifies instances of suboptimal implementations. These are subsequently optimized by the transformations' code modification part. We will only discuss the former.

DeepWeaver pointcuts consist of Prolog predicates over the Jimple intermediate bytecode representation and low-level data flow and control flow analysis results provided by the SOOT [VRCG+99] Java optimization framework.

A "`select *`" performance anti-pattern [FKI+07] is for instance characterized by code executing an expensive database query to retrieve all of a relation's columns, only to use a few specific columns afterwards. The conjunction of *DeepWeaver* conditions depicted in Figure 3.16 is given to detect such situations in its intermediate representation. The first three conditions locate the intermediate SOOT local variable `Target` which is assigned the result of the database query: a `ResultSet` instance. The invocation's argument, the query string, is expected to reside in the caller's `ExecCall` constant pool. It must be retrieved by the three subsequent conditions for it to be replaced by an optimized database query.

The optimized query only includes those columns that are actually used, identified by an auxiliary predicate `columnUsed`. It consists of the conditions listed in Figure 3.17. The first condition finds all actual uses `Use` of the `Target` variable through the SOOT-provided results of a classical reaching definitions [NNH05] analysis. The next condition verifies that the `ResultSet` is used as the receiver of a `get` method invocation. Its string argument `ColumnArg` contains the name of an operationally selected database column.

In addition to the predicates shown in Figure 3.17, *DeepWeaver* also provides access to SOOT's intraprocedural control flow graphs through predicates such as `dominates/2` and `between/4` which respectively check whether all paths to a Jimple instruction go through another and enumerate nodes reach-

```
1 reaching_def(Target, Use, false),
2 call("* ResultSet.get*(String)", Use, Location, ColumnArgs),
3 encloses(Location, Use),
4 member(ColumnArg, ColumnArgs),
5 local_constant_def(ColumnArg, Column)
```

Figure 3.17: Definition of the DeepWeaver `columnUsed` predicate that selects the names of those columns in the result of a database query that are actually used by the querying program.

able on all or any path between two others. Predicates `loop/1` and `encloses/2` together provide functionality to detect whether an instruction resides in a loop. The user is however not shielded from the details of the intermediate program representation that underlies the control flow graphs. There are moreover no readily available predicates to access SOOT's higher-level points-to and call graph analyses.

3.6 Concluding Evaluation of the Surveyed Tools

Table 3.4 evaluates the surveyed tools on the criteria for a general-purpose pattern detection tool introduced in Section 2.6. Throughout this chapter, we have briefly explained the entries in Table 3.4 on a tool-by-tool basis. We conclude our chapter by revisiting the entries for each criterion individually.

3.6.1 Evaluation on the Criteria for the Pattern Specification Language

Table 3.1 summarizes the pattern specification language of each tool.

CSL1: Supports the specification of behavioral and non-behavioral characteristics in a uniform language
Criterion **CSL1** is not fulfilled by any of the surveyed tools. METAL comes closest to fulfilling this criterion. It is tailored to control flow characteristics and has built-in support for basic data flow characteristics (e.g. variable synonym tracking). JUNGL follows METAL. It is tailored to syntactic and control flow characteristics. Its specification language is sufficiently powerful to support expressing non-syntactic characteristics in terms of syntactic characteristics. The latter is also true for the other tools marked with ±. However, expressing other characteristics in terms of syntactic characteristics is hard and error-prone (cf. Section 2.2).

CSL2: Results in descriptive pattern specifications Criterion **CSL2** is fulfilled to a large extent (±) by most of the surveyed specification languages —at least for those characteristics that can be specified. The scope of this survey is limited to declarative specification languages. Tools tailored to a single characteristic result in very descriptive specifications (+): syntactic characteristics in the case of INTELLIJ SSR and SELLINK ET AL. 'S NATIVE PATTERNS; structural characteristics in the case of 4THOUGHT, PAT, RICHNER'S VISUALISATION TOOL, CODEQUEST and JQUERY; control flow characteristics in the case of CONDATE and PATH LOGIC PROGRAMMING; data flow characteristics in the case of JTL and PQL.

CSL3: Supports expressing explicit points of variation among pattern instances Because of meta-variable provisions, the concrete syntax featuring specification languages of SCRUPLE, INTELLIJ SSR and CONDATE fulfill criterion **CSL3** at least to some extent. The other surveyed tools feature expressive means to convey that a pattern's characteristic is only one of several or anything but allowed —usually inspired by logic connectives.

CSL4: Provides means for abstraction and reuse among specifications Specification languages based on a logic or grammar formalism fulfill this criterion to the full extent. Explicitly adorning the program representation with annotations to which other specifications can refer (METAL, MJ) or defining macros (TAWK, TRANS) comprise other prevalent, but less rigorously defined (±) instances of pattern specification reuse. With the exception of CONDATE, abstraction facilities are only absent from tools that specialize in syntactic characteristics.

CSL5: Hides program representation details Specification languages that support the concrete syntax of the base program fulfill this criterion best.

The following observations about concrete syntax in specification languages can be made:

I/ The specifications in INTELLIJ SSR and SELLINK ET AL. 'S NATIVE PATTERNS comprise stand-alone fragments from the base program. Their specification language only extends the syntax from the base program with provisions for meta-variables. Neither language offers substantial abstraction facilities. Stand-alone SCRUPLE specifications and the unparsed patterns in CONDATE's specifications are relatively close to the base program's syntax, but include many non-native constructs out of a desire to influence the matching semantics and out of sheer necessity respectively. Once again, both lack substantial abstraction facilities.

II/ Three distinct approaches to combining concrete syntax with a logic-based specification language can be discerned. LOGICAJ2 and GENTL complement the ordinary terms in the logic language with the base program's concrete syntax for individual statements, declarations and expressions mixed with meta-variables. In this approach, concrete syntax is integrated within the logic language. While PQL specifications comprise only the reference-related subset of the base program's concrete syntax, it goes beyond concrete syntax for individual concrete syntax elements and associates for instance a semantics to the matching of sequences of statements. In this approach, the logic language is completely hidden and the base program's concrete syntax is extended with non-native constructs that absorb select features from the underlying logic language. Logic operators are for instance included and instead of predicate definition, procedure definition and invocation provide for reuse and abstraction in specifications. The resulting specifications still resemble valid excerpts from the base program. JTL takes a third approach which amounts to little more than a new surface syntax for the underlying logic language. These departures from the regular logic syntax are carefully chosen to ensure the resulting specification language is more familiar to the base program's implementers. The specifications for all but the simplest patterns however lack any real resemblance to base program fragments.

III/ Finally, one can observe that not a single logic-based specification language integrates the complete concrete syntax of the base program.

3.6.2 Evaluation on the Criteria for the Pattern Detection Mechanism

Table 3.2 summarizes the detection mechanism of each approach.

CDM1: Reports elements from the program's source code For tools that rely on a program representation that only carries syntactic or structural program information, **CDM1** is fulfilled as long as no reported elements stem from an advance canonicalization of this information such as the one applied by SCRUPLE. As abstract syntax trees underlie the control flow graphs employed by JUNGL and METAL, they too report elements from the program's source code. Although the program representation of PQL and JTL carries data flow information, only structural information is reported. The former shields users completely from its points-to information, while the latter supports references to its intra-procedural dependency information through scratches but does not report them since they are anonymous.

CDM2: Facilitates user assessment of reported instances Providing a ranking for results is the only means through which the surveyed tools accommodate criterion **CDM2**. The actual information conveyed by the rankings and the way in which they are determined differs greatly. The same goes for the motivation to provide the user with such a ranking in the first place. The ranking provided by the FUZZY FUJABA variant truly reflects an instance's likelihood of being a false positive and is motivated by a desire to use less precise pattern specifications that cover many implementation variants. It is computed according to the theory of fuzzy logic. The ranking provided by the column's remaining entries is not supported by a well-defined formalism. The ranking computed by METAL and the closely related MJ aims to report true bug pattern instances above likely false positives and severe bugs above more benign ones. The ranking reported by PTIDEJ reflects the importance of and the amount of specification constraints an instance does not adhere to.

CDM3: Supports *implicit* points of variation among pattern instances The third column of Table 3.2 summarizes how each detection mechanism recognizes implicit implementation variants. Note that the entries in this column are scarce. Strictly spoken, SCRUPLE and PQL owe this ability to their program representation rather than their detection mechanism. The points-to analysis of the latter automatically recognizes syntactically differing expressions that might evaluate to the same object at run-time. The representation of all loop constructs in the former's program representation is canonicalized in advance. GRAPSR's canonicalizations are in contrast performed on-the-fly by the detection mechanism. The tool relies on the tolerant nature of its data flow representation as well, although it is very basic compared to the one of PQL. SELLINK ET AL. 'S NATIVE PATTERNS perform on-line canonicalizations expressed as user-specified, problem-specific rewrite rules. However, it lacks predefined language-specific rules. METAL has more advanced online provisions. It tracks variable synonyms induced by simple assignments and simplifies boolean expressions. Under the assumption that mismatches originate from distorted instances of a pattern, PTIDEJ takes more radical

measures by automatically relaxing constraints. In the evaluation table, tools marked with a + symbol perform on-line canonicalizations comprising both predefined language-specific ones and any additional problem-specific ones defined by the user. The entry for the FUZZY FUJABA variant comprises a − symbol as it does not perform any canonicalizations.

CDM4: Can be extended with user-defined search strategies Not a single surveyed tool explicitly supports the definition of user-defined search strategies. In Chapter 5, we will describe and subsequently argue against the options available to users of logic meta programming tools. Users of other tools have little choice but to alter the existing implementation of the detection mechanism in a language that differs significantly from the specification language. To alter SCRUPLE's detection mechanism, for instance, knowledge is required about the code pattern automatons it relies on. Although conceptually simpler, altering the translation of PQL specifications to Datalog still requires knowledge of the Java visitors that implement it.

Based on the overview in Table 3.2, the following observations about the surveyed detection mechanisms can be made:

I/ General-purpose proof procedures for logics constitute the bulk of the detection mechanisms' pattern search strategies —no surprise given the prevalence of logic-based specification languages illustrated by Table 3.1. These range from tabled or plain resolution for Prolog over model checkers for temporal formulae to various evaluation techniques for Datalog. ASTLOG on the other hand employs a variant of resolution that is well-suited to AST traversals.

II/ Algorithms specifically tailored to a particular pattern characteristic can only be discerned in the third row of the table which groups tools tailored to control flow characteristics. These algorithms are implemented in the declarative as well as in the imperative programming paradigm. As pointed out in the previous section, the syntax of the logic-based tools in the lower row deviates significantly from the one their formalism is usually associated with. In the case of *PQL*, one could also state that the logic program *PQL* specifications are translated to implements an algorithm that hides the intricate details of the data flow representation.

III/ Another observation is that the particular detection mechanism a tool employs is largely fixed by the tool's implementers. Apart from the interactive pattern search strategies through which FUZZY FUJABA and GRASPR solicit the user for advice on intermediate results, all other provisions for control over the detection mechanism require either annotations or invasive changes to pattern specifications. Neither changing whether instances of a pattern are to be found across function boundaries in METAL, nor changing whether a meta-variable should bind a single rather than all of a method's parameters in INTELLIJ SSR requires invasive changes to the essence of a pattern specification. Users of SCRUPLE and CONDATE are however forced to pollute the concrete syntax of their specifications with non-native constructs to control the detection mechanism.

3.6.3 Evaluation on the Criteria for the Program Representation

Table 3.3 summarizes the program representations of the surveyed tools.

CPR1: Includes behavioral and non-behavioral program information explicitly
Criterion **CPR1** is not fulfilled by any of the surveyed tools. METAL and DEEPWEAVER are close (±) to fulfilling this criterion. The nodes in METAL's inter-procedural control flow graphs stem from an abstract syntax tree. This information is complemented by basic data flow information computed by the detection mechanism to handle variable synonyms and function parameters. DEEPWEAVER is the only tool that offers basic structural, intra-procedural control flow and intra-procedural data flow information. However, DEEPWEAVER users are exposed to their intricate details and to the intermediate representation that underlies the control flow and data flow information. Syntactic information is moreover missing.

From the overview Table 3.3, we furthermore observe the following about the surveyed tools:

I/ A quick glance immediately reveals that most approaches settle on a representation that exclusively carries either syntactic, structural, control flow or data flow information. Few entries complement the table's diagonal. Notable exceptions are the representations for programs in the object-oriented paradigm which virtually always include basic structural information about the program's classes and methods.

II/ A second observation is that the program representation underlying the control flow and data flow representations almost always constitutes an intermediate or bytecode representation. Only the instructions in the control flow graphs of *Metal* and *JunGL* stem from the program's abstract syntax trees. Not a single approach offers data flow information with respect to a program's abstract syntax tree nodes.

III/ Thirdly, the majority of pattern detection tools offering control flow information restrict themselves to the intra-procedural case —*Metal* being the exception. *PQL* constitutes the only tool offering non-trivial inter-procedural data flow information.

Table 3.1: Overview of declarative pattern specification languages employed by surveyed tools.

Approach	In a Nutshell	Abstraction Facility	Representation Exposure
SCRUPLE	base language's concrete syntax extended with non-native constructs that influence matching semantics	-	low
ASTLOG	Prolog query against an implicit AST traversal node	predicate definition	high
Fuzzy FUJABA	visual graph rewrite rule weighted by its expected precision	rule definition	high
Centaur	Prolog query against logic term constructed from s-expression with call-outs to Lisp	predicate definition	high
IntelliJ SSR	subset of base language's concrete syntax with GUI-specified constraints on meta-variables that influence matching semantics	-	low
Declarative subset of PMD	XPath query against XML document	-	high
TAWK	regular abstract syntax tree expressions	textual macro expansion	high
LogicAJ2 and GenTL	Prolog query with concrete syntax conditions	predicate definition	medium
Sellink et al. 's Native Patterns	concrete syntax with meta-variable names corresponding to non-terminal names in the base language's grammar	-	low
PTIDEJ	Claire constraint satisfaction problem with weighted constraints	(low-level) constraint definition	high
4Thought	visual GraphLog query	rule definition	high
Pat, Richner's visualisation tool, jQuery	Prolog query	predicate definition	high
Metal	deterministic finite state machine with concrete syntax transition guards and call-outs to C	inter-machine through AST annotations	low
Condate	constrained reachability query between nodes identified by unparsed patterns comprising concrete syntax subset with meta-parentheses	-	medium
MJ	*see Metal*, only subset of base language's concrete syntax	*see Metal*	medium
TRANS	computational logic formula over graph labels within rewrite rule	macro definition	high
Path Logic Programming	Prolog query with embedded regular path expressions over graph labels	predicate definition	high
JunGL	Datalog query with embedded regular path expressions as stream comprehension within functional transformation language	edge constructor definition	high
JTL	Datalog query in Java-like syntax featuring set quantification abstractions	predicate definition	medium
GRASPR	graph grammar production rule transcribed as s-expression	rule definition	high
PQL	reference-related subset of base language's concrete syntax extended with logic operators translated to Datalog query	procedural abstraction	low
DeepWeaver	Prolog query	predicate definition	extremely high

Table 3.2: Overview of detection mechanisms employed by surveyed tools.

Approach	Search Strategy	User Steering	Implicit Variation Points	Instance Ranking
SCRUPLE	code pattern automaton	wildcard suffixes control nesting depth	advance loop construct canonicalization	-
ASTLOG	resolution against implicit current object	-	-	-
Fuzzy FUJABA	bottom-up and top-down scheduled rules	interaction on intermediate results	-	applied rules weighted by expected false positives
Centaur	Prolog resolution	co-routined evaluation with Lisp	-	-
IntelliJ SSR	*closed source*	matching constraints on meta-variables	-	-
Declarative subset of PMD	XPath evaluation	-	-	-
TAWK	code pattern automaton	-	-	-
LogicAJ2, GenTL	Prolog resolution	-	-	-
Sellink et al. 's Native Patterns	term rewriting	-	problem-specific equivalences between native patterns	-
PTIDEJ	explanation-based CSP solver	-	automatic constraint relaxation	relaxed constraints weighted by importance
4Thought	GraphLog resolution	-	-	-
Pat, Richner's visualisation tool	Prolog resolution	-	-	-
CodeQuest	Datalog evaluation through translation to SQL	-	-	-
jQuery	tabled Prolog resolution	-	-	-
Metal	state machine simulation over control flow graph	declaration determines interp. or intrap. simulation	variable synonym tracking, boolean expression simplification	heuristics favouring low inspection effort and high historical adherence
Condate	constrained version of Liu et al. [LRY+04]	(often necessary) meta-parentheses	-	-
MJ	*see Metal*	-	-	*see Metal*
TRANS	CTL model checker with BDD valuation representation	-	-	-
Path Logic Programming	tabled Prolog resolution and de Moor et al. [dLW03]	-	-	-
JunGL	ML, Datalog evaluation and de Moor et al. [dLW03]	-	-	-
JTL	mixed top-down & bottom-up Datalog evaluation	-	-	-
GRASPR	bottom-up chart parser featuring chart monitor	monitor-triggered user advice	monitor-triggered canonicalizations, syntactically differing data flow origins	-
PQL	Datalog evaluation through BDD manipulation	-	syntactically differing reference expressions	-
DeepWeaver	Prolog resolution	-	-	-

Table 3.3: Overview of program representations employed by surveyed tools.

Approach	Base Language	AST	Structural Information	Control Flow	Data Flow
SCRUPLE	C, PL/AS	•	-	-	-
ASTLOG	C/C++	•	-	-	-
Fuzzy FUJABA, LogicAJ2, GenTL	Java	•	oo entities and relations	-	-
Centaur	user-defined	•	-	-	-
IntelliJ SSR, Declarative subset of PMD	Java	•	-	-	
TAWK	C, MUMPS	•	-	-	-
Sellink et al. 's Native Patterns	COBOL	•	-	-	-
PTIDEJ	Java	-	oo entities and relations	-	-
4Thought	-	-	provided by user	-	-
Pat	C++	-	oo entities and relations	-	-
Richner's visualisation tool	Smalltalk	-	oo entities and relations	-	-
CodeQuest, jQuery	Java	-	oo entities and relations, field r/w and invocations in method	-	-
Metal	C	•	-	interp. on AST	-
Condate	C	-	-	intrap. on IR	-
MJ	Java	-	-	intrap. on IR	-
TRANS	Java, C	-	-	intrap. on IR	-
Path Logic Programming	.NET IL	-	-	intrap. on IR	-
JunGL	C#	•	-	lazily constructed intrap. on AST	-
JTL	Java	-	basic oo	-	intrap. value dependency on bytecode
GRASPR	Lisp	-	-	control environments encoded in data flow	interp. value dependency on primitives
PQL	Java	-	basic oo, context-sensitive call graph	-	interp. context-sensitive points-to analysis on IR
DeepWeaver	Java	-	basic oo	intrap. on IR	intrap. value dependency on IR

Table 3.4: Evaluation of surveyed tools on general-purpose pattern detection criteria.

	CSL1	CSL2	CSL3	CSL4	CSL5	CDM1	CDM2	CDM3	CDM4	CPR1
SCRUPLE	.	±	±	.	+	.	.	±	.	.
ASTLOG◇, Centaur◇	±	±	+	+	.	+	.	.	.	.
Fuzzy FUJABA	±	±	+	+	.	+	+	.	.	.
IntelliJ SSR	.	+	±	.	+	+	.	.	.	.
Declarative subset of PMD	.	±	+	.	.	+	.	.	.	.
TAWK	.	±	+	±	.	+	.	.	.	.
LogicAJ2◇, GenTL◇	±	±	+	+	+	+	.	.	.	.
Sellink et al. 's Native Patterns	±	+	+	.	+	+	.	±	.	.
PTIDEJ	.	±	+	+	.	+	+	.	.	.
4Thought◇, Pat◇, Richner's visualisation tool◇, CodeQuest◇, jQuery◇	.	+	+	+	.	+	.	.	.	.
Metal	±	±	+	±	+	+	+	±	.	±
Condate	.	+	±	.	±	.	.	.	.	.
MJ	.	±	+	±	+	.	+	.	.	.
TRANS	.	±	+	±	.	.	.	.	.	.
Path Logic Programming◇	.	±	+	+	.	.	.	.	.	.
JunGL	±	±	+	+	.	+	.	.	.	.
JTL◇	.	+	+	+	±	+	.	.	.	.
GRASPR	.	±	+	+	.	.	.	+	.	.
PQL◇	.	+	+	+	+	+	.	±	.	.
DeepWeaver◇	.	±	+	+	.	.	.	.	.	±

CHAPTER 4

AN EXAMPLE-DRIVEN APPROACH TO PATTERN DETECTION

This chapter outlines our primary contribution to pattern detection: an example-driven approach that fulfills all of the criteria for a general-purpose pattern detection tool. We introduce and motivate each cornerstone of this approach. As the cornerstones are complementary, we clarify their contributions to our approach in terms of the criteria they help to fulfill. We illustrate these contributions by applying a concrete instantiation of each cornerstone, detailed in subsequent chapters, separately to a running example.

4.1 Cornerstones of the Approach

This chapter outlines the cornerstones of our example-driven approach to pattern detection. *Logic meta programming* lends its machine-executable proof procedure for logic formulas, providing our approach with the potential to separate the specification of patterns from the search for their instances. This founding cornerstone is introduced in Section 4.2. Two cornerstones of our approach ensure that this potential is fully realized. The descriptiveness of the resulting specifications sets our approach apart. The first, *example-based specifications*, incorporates source code excerpts within logic formulas. This cornerstone is introduced in Section 4.3. The second, a *domain-specific unification* procedure, incorporates whole-program analysis results in the comparison of individual program elements. It is introduced in Section 4.4. Through its quantification of truth, *fuzzy logic* facilitates user assessment of the detected pattern instances. This cornerstone is introduced in Section 4.5. The final cornerstone, *open implementation*, crosscuts the other cornerstones to ensure their user-extensibility. It is introduced in Section 4.6.

Figure 4.1 depicts a detailed architectural overview of the research prototype that instantiates our example-driven approach. Instantiated cornerstones are depicted as puzzle pieces. Chapter 5, Chapter 6 and Chapter 7 detail the concrete instantiations of the logic meta programming, domain-specific unification and example-based specification cornerstones. The concrete instantiation of the fuzzy

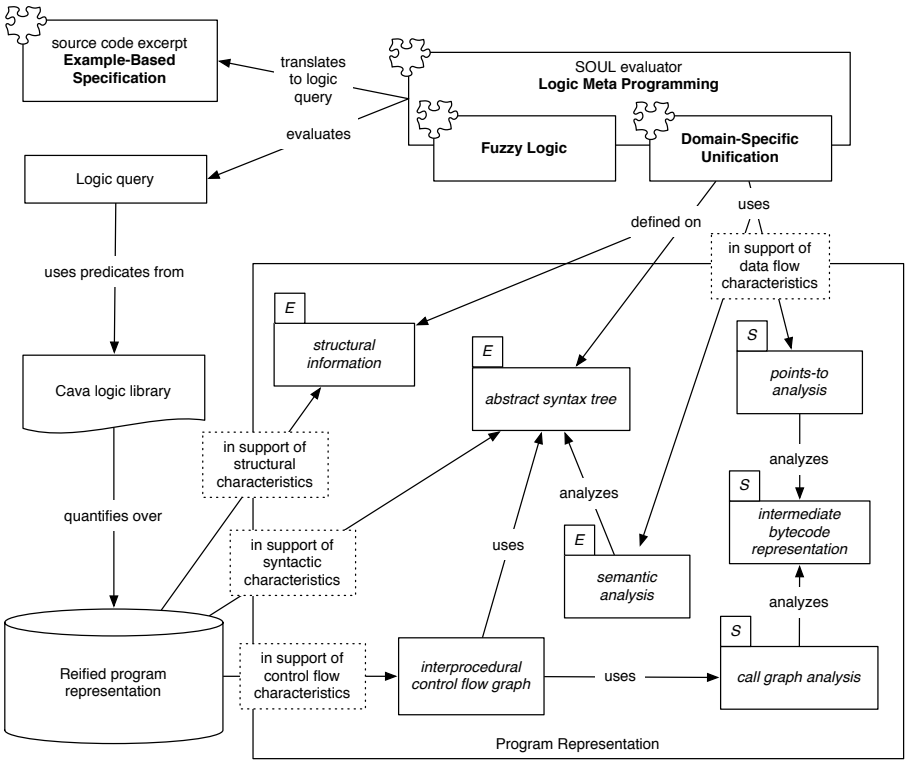


Figure 4.1: Architectural overview of a concrete instantiation of our approach.

logic cornerstone is discussed alongside the logic meta programming cornerstone. The instantiation of the open implementation cornerstone is revisited in each chapter as it crosscuts the concrete instantiation of each cornerstone. We discuss these instantiations here as well, but only to the extend necessary to illustrate their respective cornerstones on a running example.

Our example-driven approach fulfills all of the criteria for a general-purpose pattern detection tool identified in Section 2.6. For each cornerstone, Table 4.1 lists the individual contributions to our approach in terms of the criteria it helps to fulfill. The entries in the table are explained in the remainder of this chapter. Entries of the form + indicate that the cornerstone in the column header contributes to the fulfillment of the criterion in the row label. Entries of the form CPR1 indicate that whether or not a cornerstone helps our approach fulfill a criterion, depends on whether criterion CPR1 is fulfilled.

Running Example

We will use the well-known *getter method* best practice pattern [Bec96] as a running-example. The pattern consists of a method that returns the value of an instance variable. Accessing this variable through the getter method rather than directly eases evolution. Clients of a class are shielded from changes to its internals. Method `getVar()`, depicted in Figure 4.2, corresponds to the prototypical Java implementation of the pattern.

	Logic Meta Programming	Example-Based Specification	Domain-Specific Unification	Fuzzy Logic	Open Implementations	
CSL1	CPR1	+	+	.	CPR1	Supports the specification of behavioral and non-behavioral characteristics in a uniform language
CSL2	.	+	+	.	.	Results in descriptive pattern specifications
CSL3	+	.	.	.	.	Supports expressing explicit points of variation among pattern instances
CSL4	+	.	.	.	.	Provides means for abstraction and reuse among specifications
CSL5	.	+	+	.	.	Hides program representation details
CDM1	¬ CPR1	+	.	.	.	Reports elements from the program's source code
CDM2	.	.	.	+	.	Facilitates user assessment of reported instances
CDM3	.	+	+	.	.	Supports implicit points of variation among pattern instances
CDM4	.	.	.	.	+	Can be extended with user-defined search strategies
CPR1	.	.	.	.	.	Includes behavioral and non-behavioral program information explicitly

Table 4.1: Overview of the pattern detection criteria that each cornerstone of our example-driven approach helps to fulfill.

```
1 class Y {
2   private X var;
3   public X getVar { return var; }
4   public void setVar(X val) { var = val; }
5 }
```

Figure 4.2: Prototypical implementations of the getter and setter method best practice patterns.

The consistent use of the pattern can be enforced through a pattern detection tool. There should be no direct accesses to a variable protected by a getter method. However, despite the pattern's simplicity, supporting the detection of its implementation variants using a descriptive specification is hard.

4.2 Cornerstone: Logic Meta Programming

We initiate the discourse on our example-driven approach to pattern detection with a discussion of its logic meta programming roots.

Logic formulas can be used in a straightforward manner to specify a pattern. This merely requires a reification of the program representation such that variables can range over its elements. Executing a proof procedure will establish whether program elements exhibit the characteristics specified in a formula.

Machine-executable proof procedures range from tabled [RC97, CW96] or plain

```
?methodDeclaration getsFragmentNamed: ?fieldName
                   ofFieldDeclaration: ?fieldDeclaration
                   in: ?typeDeclaration if
    ?methodDeclaration isMethodDeclaration,
    ?fieldDeclaration fieldDeclarationHasFragments: ?fragments,
    [?fieldDeclaration parentTypeDeclaration] equals: ?typeDeclaration,
    ?typeDeclaration equals: [?methodDeclaration parentTypeDeclaration],
    ?fragments contains: ?fragment,
    ?fragment variableDeclarationFragmentHasName: ?fieldName,
    ?fieldName simpleNameHasIdentifier: ?fieldIdentifier,
    ?methodDeclaration methodDeclarationHasBody: ?block,
    ?block blockHasStatements: ?statements,
    [?statements size = 1],
    ?statements contains: ?statement,
    ?statement returnStatementHasExpression: ?expression,
    ?expression simpleNameHasIdentifier: ?expressionIdentifier,
    ?fieldIdentifier equals: ?expressionIdentifier
```

Figure 4.3: SOUL rule for the prototypical implementation of the getter method in Java.

[Rob65] resolution for Prolog [EK76] over model checkers for temporal formulas to various evaluation techniques for Datalog [CGT89]. Powerful pattern detection tools result from procedures that give rise to a Turing-complete specification language. In this case, pattern specifications can be considered logic meta programs.

Logic meta programming (LMP) refers to the use of a logic program to manipulate other programs.[1] The declarative nature and expressiveness of logic programs facilitates their use as descriptive specifications of a pattern's characteristics rather than the search for its instances. However, nothing precludes users from implementing an operational search themselves.

We will identify two shortcomings of logic meta programming that lead to operational and convoluted pattern specifications: quantification over and unification of reified program representation elements. We will remedy these problems by adopting the example-based specification and domain-specific unification cornerstones in addition to the founding logic meta programming cornerstone.

4.2.1 Running Example Revisited

We revisit the running example to illustrate logic meta programming in isolation from the other cornerstones of our approach.

Figure 4.3 depicts a logic meta programming specification for the prototypical implementation of the getter method shown in Figure 4.2. The rule expresses what it means for a method declaration *?methodDeclaration* to declare a getter method for one of the variable declaration fragments[2] named *?fieldName* of a field declaration *?fieldDeclaration* in the type declaration *?typeDeclaration*. As such, it serves as a specification of the getter method.

The syntax[3] of the depicted rule stems from the SOUL logic programming lan-

[1] We do not restrict the term to logic programs that manipulate parts of themselves using the meta-level functionality of a logic programming language.

[2] Note that Java allows multiple fields of the same type to be declared in one declaration.

[3] In Prolog, the head of the rule would read as `getsFragmentNamedOfFieldDeclarationIn(MethodDeclaration, FieldName, FieldDeclaration, TypeDeclaration):-`. The syntax for a

guage [Wuy98, Wuy01, Sou08]. SOUL is the Prolog [EK76] and Smalltalk [GR83] hybrid used as instantiation of the logic meta programming cornerstone in our research prototype. SOUL programs comprise both logic conditions (e.g. line 12) and Smalltalk expressions (e.g. line 13). Objects are exchanged transparently through logic variables as if they were ordinary values in either language.

Other specifications for the getter method are possible, but this one illustrates the *hybrid language* characteristic of SOUL. As mentioned in the introduction, logic meta programming requires a reification of the program representation such that variables can range over its elements. This reification is provided by the predicates in the CAVA library (cf. Section 5.2). For instance, the predicate `isMethodDeclaration/1` in the first condition of the rule binds its argument to a method declaration. The hybrid language characteristic of SOUL allows us to forgo the prevalent transcription to compound terms (e.g. `methodDeclaration(?modifiers,...,?body)`). The reified version of the method declaration is the declaration itself (i.e. an abstract syntax tree) which can be queried by sending messages to it. For instance, we use a *Smalltalk term* (i.e. a Smalltalk expression extended with logic variables and demarcated by square brackets) on line 7 to retrieve the type declaration in which the method is declared.

The first condition of the rule binds its variable to a method declaration. The second condition provides a binding for a field declaration and its variable declaration fragments. The Smalltalk terms on the left- and right-hand sides of the `equals:/2`[4] conditions express that the getter method and the field it protects must be declared in the same type. Lines 8 through 10 access the strings that identify each fragment of the field declaration. Lines 11 through 14 state that the method declaration must have a body that consists of a single statement. The condition on line 15 requires this statement to be a return statement and provides a binding for the expression it returns. Finally, the last condition ensures that this expression is a simple name that is named after the string that identifies the variable declaration fragment.

4.2.2 Motivation for the Logic Meta Programming Cornerstone

We briefly clarify the entries in the first column of Table 4.1. These consist of the contributions of the logic meta programming cornerstone in terms of the criteria it helps our approach to fulfill.

Provides Support for Expressing Explicit Points of Variation

Logic formalisms are particularly suited for expressing variation among the instances of a pattern (criterion **CLS3**) using logic connectives (e.g. conjunction, disjunction and negation). Logic variables can be used to relate aspects of multiple characteristics across a specification. There are, for instance, multiple occurrences of the *?methodDeclaration* variable in the specification of the getter method. The bindings for these occurrences are consistent and only differ per pattern instance.

predicate in SOUL closely resembles the one of Smalltalk for a message sent to the first argument of the predicate. Logic variables are preceded by a question mark. Section 5.1.1 discusses the syntax and semantics of SOUL in more detail.

[4] The logic fact `?x equals: ?x.` implements the `equals:` predicate which hence serves as a substitute for the `=/2` operator of Prolog.

Logic variables can also be used to indicate that an aspect of an individual characteristic is unimportant (i.e. anonymous variable).

The logic meta programming cornerstone lends our *specification language* support for expressing *explicit* points of variation among pattern variants. The domain-specific unification cornerstone will lend our *detection mechanism* support for detecting *implicit* points of variation among the variants. The latter represent different implementations of the same characteristic, while the former represent variations among the characteristics that should be detected.

The example-based specification cornerstone will adopt the logic variables from this cornerstone as its primary means to express variation. Logic meta programming will provide its logic connectives and abstraction facilities to compose example-based specifications.

Provides Abstraction and Reuse Facilities

Implication in logic formalisms and predicate definition in logic programming can be used to abstract multiple characteristics into a single characteristic. The resulting characteristic can be reused without exposing the inner details it comprises (criterion **CSL4**).

The SOUL rule for the getter method defines a predicate `getsFragmentNamed:ofFieldDeclaration:in:/4` which can be used in the body of other rules. The predicate can, for instance, be used in another rule to verify that the detected method adheres to the naming convention for getter methods.

Defining an additional rule with the same head amounts to providing an alternative specification for the same pattern. For instance, one that uses logic predicates exclusively and one that uses example-based specifications exclusively.

Provides Support for Behavioral and Non-Behavioral Characteristics

It depends on the program representation employed by its concrete instantiation whether the logic meta programming cornerstone supports the specification of syntactic, structural, control flow and data flow characteristics.

Non-syntactic characteristics can be expressed in terms of syntactic characteristics. As syntactic characteristics are only supported by abstract syntax trees, they must be included in the program representation. However, relying on syntactic characteristics to express other characteristics results in convoluted specifications with idiomatic recurring parts (see Section 2.5).

This can be avoided by providing a library of characteristics accompanied by implementations of the search for program elements that exhibit them. Logic meta programming tools can provide a predefined library of logic rules. Without such a library, the contribution of logic meta programming to criterion **CSL1** is limited to the characteristics that can be expressed by referencing the reified program representation directly. For instance, in a representation that explicitly indicates which pairs of instructions are executed consecutively, an existential quantification over this information suffices to specify a pattern of two consecutively executed instructions. This option, on the other hand, exposes users to the details of the program representation *and* its reification (cf. criterion **CSL5**).

Our query-by-example approach to pattern detection *includes both behavioral and non-behavioral information* in the program representation. However, as shown in Figure 4.1, the logic meta programming cornerstone *only reifies abstract*

```
1 ?root isAncestorOf: ?directSubclass if
2   ?directSubclass isSubClassOf: ?root.
3 ?root isAncestorOf: ?indirectSubclass if
4   ?indirectSubclass isSubClassOf: ?parent,
5   ?root isAncestorOf: ?parent

6 if ?superclass isAncestorOf: [SmallInteger]
7 if [Object] isAncestorOf: [SmallInteger]
8 if [Object] isAncestorOf: ?subclass
9 if ?superclass isAncestorOf: ?subclass
```

Figure 4.4: SOUL rules describing the ancestor relation between two Smalltalk classes.

syntax tree nodes[5] in compliance with criterion **CDM1**. By *forgoing the transcription to compound terms*, users are not exposed to the details of an arbitrary reification. SOUL's hybrid language characteristics allows the nodes to be queried using message sends (e.g. line 7 of the getter rule). The example-based specification cornerstone will hide the remaining details of these nodes and support the expression of behavioral information.

Potential for Descriptive Pattern Specifications

Using a logic language for meta programming purposes has several advantages [CH87, Wuy01]. Logic programming languages feature advanced pattern matching abilities, built-in support for non-determinism, logic connectives and powerful programming concepts such as recursion and backtracking.

Consider the problem of finding all ancestors of a particular class. As shown in Figure 4.4, it suffices to describe what it means for one class to be an ancestor of another class in logic programming. The first rule expresses that a class *?root* is the ancestor of a class *?directSubclass* if the latter is a subclass of the ancestor class. The second rule expresses that a *?root* class is also the ancestor of a subclass of a class it is already the ancestor of.

The specification of the problem is very descriptive. Logic rules describe relations between their arguments in a declarative instead of an operational manner. The ancestor rule, for instance, can be used to find all superclasses of a given class (line 6), but also to find all subclasses of a given class (line 8). Each problem requires a separate algorithm in the imperative paradigm.

The ancestor rule in Figure 4.4 illustrates the potential descriptiveness of logic rules. In contrast, the rule for the getter method in Figure 4.3 illustrates the operational nature logic meta programming specifications often incur in practice.

As mentioned in the introduction, we identify two shortcomings of logic meta programming that lead to such operational and convoluted pattern specifications: quantification over and unification of reified program representation elements.

Quantification-Related Shortcomings of LMP

We discuss the quantification-related causes for the operational nature of LMP specifications first. They will be remedied by the example-based specification cor-

[5] The structural information consists of select AST nodes, while the control flow graph referred to by example-based specifications is constructed on top of the AST.

```
1 foundAssignment(?context,assign(?var,?value)).
2 processAssignment(?context,assign(?var,?value),<?var,?value>).

3 methodWithAssignment(?method,assign(?var,?value)) if
4   traverseMethodParseTree(?method,<?var,?value>,
5                           foundAssignment,processAssignment)
```

Figure 4.5: SOUL traversal of the AST for a Smalltalk method to search for assignments.

nerstone of our approach. The unification-related causes will be described along with the domain-specific unification cornerstone that remedies them.

Structural program information (see Section 2.5.2) concerns the organisation of a program in terms of the relations (e.g. inheritance) between key program entities (e.g. classes). Logic meta programming tools can support expressing structural characteristics in a straightforward manner. This merely requires reifying structural relations as predicates over the reified program entities in the relation. The *relational nature of logic programming facilitates quantifying over the reified relations.* This, in turn, results in descriptive specifications of structural characteristics. The descriptive ancestor rule of Figure 4.4, for instance, quantifies existentially over an inheritance relation reified as the multi-directional `isSubClassOf:/2` predicate.

Logic meta programming tools can support syntactic information in a similar manner by reifying the hierarchical relations between abstract syntax tree nodes. However, expressing syntactic characteristics (cf. Section 2.5.1) in terms of these relations is less straightforward. This requires an operational traversal of an AST. Depending on the employed reification of the AST[6], the traversal might require an on-the-fly reconstruction of the tree in case its reification had it flattened into multiple compound terms. To shield the user from these operational traversals, logic meta programming tools can provide higher-order predicates that implement a generic traversal.

In SOUL, for instance, the higher-order predicate `traverseMethodParseTree(?method,?result,?found,?process)` traverses a Smalltalk AST while applying predicate *?process* to every node for which predicate *?found* succeeds. Figure 4.5 depicts a SOUL program, in traditional predicate syntax, that can be used to find assignments in a Smalltalk method. A *?context* argument, describing the context in which the current traversal node resides (e.g. the path followed through the AST), is passed to each predicate. This is often necessary to express syntactic characteristics. To find expressions that use the assignment itself as a value, for instance, the parents of the assignment need to be taken into account.

Compared to non-LMP tools specialized in *syntactic characteristics* (see Section 3.2), the *resulting specifications are operational rather than descriptive.* Traversal contexts and higher-order predicates only add to their complexity.

[6] The intricacies of reification should not be underestimated. Consider the abstract syntax tree with root `a`. Root node `a` has two children `b` and `c`, while node `c` has a child `d`. For nearly all of the surveyed LMP tools, reification amounts to a transcription into compound terms. In case the whole AST is mapped onto a single compound term (e.g. `node(a,<node(b,<>),node(c,<node(d,<>)>))`), users have to perform a tree traversal of the compound to check that `a` is an ancestor of `d`. In case the tree is flattened into multiple compound terms (e.g. `child(a,b)`, `child(a,c)` and `child(c,d)`), users have to reconstruct the tree to check the ancestor relation between `a` and `d`. This is common in formalisms that are compound-free.

Having users quantify over reified control flow graphs to express control flow characteristics is also possible, but poses similar problems. In addition, *cycles in control flow graphs may cause termination problems.* Having users quantify over reified data flow information causes similar problems, but in addition exposes them to the *intricate details of the intermediate program representations* they are usually computed for (see Section 2.5.4). We will illustrate these problems in Section 5.3.4 by means of Figure 5.13, but they can already be discerned among the DEEPWEAVER rules depicted in Figure 3.16 and Figure 3.17 of our survey.

4.2.3 Concrete Instantiation in Brief

Chapter 5 details the concrete logic meta programming instantiation in the research artifact used to validate our approach: the Smalltalk-Prolog hybrid [Wuy98, Wuy01, Sou08] and our CAVA library of predicates for reasoning about Java. Here, we briefly outline the information in its program representation as depicted in the architectural overview of Figure 4.1.

The predicates in the CAVA library reify structural and syntactic information from the Java model and DOM of the Eclipse JDT Core Component [Ecl08a] respectively (cf. Figure 2.3 and Figure 2.2).

To comply with criterion **CPR1**, our program representation also includes semantic analysis results from the Eclipse JDT Core Component [Ecl08a] and the context-insensitive points-to analysis results from the SPARK [Lho02] component of the SOOT Java Optimization Framework [VRCG+99] respectively (cf. Figure 2.6). These are whole-program analyses that take the code of the whole program into account. An intra-procedural must-alias analysis is included as well. It offers definite aliasing information about expressions within a single method.

However, to comply with criterion **CDM1**, we do not reify the program analyses information. Otherwise, users would be exposed to their details —including the JIMPLE intermediate representation for which the points-to analysis is computed (cf. Figure 2.4). Instead, the analyses will be used by the instantiation of the *domain-specific unification* cornerstone in the comparison of reified program elements.

An inter-procedural control flow graph is computed on-the-fly whenever its reification is requested by SOUL. The nodes in this graph stem from the Eclipse ASTs. Its computation relies on the points-to analysis results to resolve late binding in method invocations. While the CAVA library includes predicates to query this graph, the *example-based specification* cornerstone will provide a more descriptive means to express control flow characteristics.

The hybrid language characteristic of SOUL is not crucial to the logic meta programming cornerstone of our approach. However, it opens the door to the imperative paradigm in specifications wherever it is more convenient. More importantly, we do not have to reify the information in the program representation as compound terms. Instead, the reified version of an Eclipse AST node is the AST node itself.

We rely on a linguistic symbiosis [GWDD06] between SOUL and Java. It is established transitively by combining the existing symbiosis between SOUL and Smalltalk with the symbiosis between Smalltalk and Java provided by the JAVACONNECT [Jav] library. We refer to Figure 5.1 for an illustration of this symbiosis. For now, it suffices to mention that it enables quantifying over any object that is reachable in the Smalltalk run-time image —including Smalltalk objects that function as

```
1  if jtStatement(?statement){ return; }

2  if jtStatement(?statement){ return ?expression; }

3  if jtStatement(?statement){ return ?expression; },
4     ?expresion isExpression

5  if jtStatement(?statement){ return ?expression; },
6     not(jtStatement(?statement){ return; })
```

Figure 4.6: Example-based specifications embedded in SOUL queries.

proxies for Java objects stemming from an Eclipse instance.

Relying on linguistic symbiosis for the reification renders reconstituting the actual AST nodes from their reified counterparts trivial at any point in the proof procedure. In particular, incorporating analyses in the unification of abstract syntax tree nodes is facilitated. The context within the program of each node can be obtained easily (cf. line 7 of Figure 4.3).

4.3 Cornerstone: Example-Based Specification

This section introduces the second cornerstone of our approach. It is detailed in Chapter 7. The cornerstone adopts source code excerpts in the *concrete syntax* of the program under investigation as example-based specifications. An example-based specification of a pattern corresponds to the *prototypical implementation* of its essential machine-verifiable characteristics.

Figure 4.2 depicts the prototypical implementation of the getter method. It is prototypical in the sense that its instructions implement only the essential characteristics of the pattern. Real-world implementations often include instructions that log accesses to the field or initialize the field on the first invocation of the method. The code exemplifies the essential characteristics shared by the pattern's instances: they *return* the value of the *private* field they protect and are *named* after.

Template Terms in Logic Meta Programming Specifications

A pattern specification language should support expressing explicit points of variation among the instances of a pattern (criterion **CSL3**). Likewise, means for abstraction and reuse among specifications should be provided (criterion **CSL4**). On these aspects, the example-based and logic meta programming cornerstones of our approach complement each other (cf. Table 2.1). We therefore embed source code excerpts in logic meta programming specifications. Template terms, containing source code excerpts, are introduced in the logic language. As resolvable terms, template terms can be used as conditions in the body of a rule or a query.

Figure 4.6 depicts a number of template terms for Java statements. They are embedded within logic queries[7]. Each template term comprises a single-argument predicate `jtStatement/1` in classical predicate logic notation, followed by a sequence of concrete syntax elements delimited by braces. As the predicate name indicates[8], the concrete syntax is recognized as a Java statement.

[7] Recall that *if* is equivalent to `:-` in Prolog.

[8] The prefix `jt` differentiates Java templates from Smalltalk templates.

```
1  if jtClassDeclaration(?classDeclaration) {
2        class ?className {
3          private ?fieldDeclarationType ?fieldName;
4          ?modifierList ?returnType ?methodName(?parameterList) {
5             return ?fieldName;
6          }
7        }
8     }
```

Figure 4.7: Example-based specification for the prototypical implementation of the getter method.

Upon backtracking over the template term in the first query of Figure 4.6, SOUL will present bindings for the logic variable *?statement* that match the Java source code between the braces of the template. The fourth query illustrates that templates are resolvable terms in SOUL.

Concrete Syntax in Template Terms

By adopting the concrete syntax of the base program for template terms, this cornerstone overcomes the hurdle of unfamiliar specification languages. Logic meta programming tools, in contrast, expose application programmers to an arbitrary reification of their program representation.

However, some departures from the concrete syntax are necessary. To indicate points of variation, logic variables are introduced in template terms. The template term in the second SOUL query of Figure 4.6 illustrates that logic variables can be used as placeholders for productions originating from a non-terminal in the `Statement` grammar production rule. Solutions to this query will have a return statement bound to the *?statement* logic variable, while the expression part of this statement will be bound to the *?expression* logic variable. The third logic query comprises the same template term with an additional logic condition from the CAVA predicate library. It explicitly checks that *?expression* is bound to an expression AST node.

This technique is common among the tools that feature concrete syntax surveyed in Chapter 3(i.e. [Pau92, SV98, ECCH00, BE03, RKA06, AK07, Mos05, Vol06b, Liv06]). Concretely, the grammar of the base programming language is extended with meta-variables (e.g. logic variables) that can be used in specifications as placeholders for productions that originate from a non-terminal. It is also common to introduce non-native syntax in source code excerpts as more expressive means to indicate points of variation (e.g. the negation operator `![]` in the excerpt `return ![null]`). To ensure the descriptiveness of example-based specifications (criterion **CSL2**), our approach limits concrete syntax departures to the bare minimum.

4.3.1 Running Example Revisited

Figure 4.7 depicts an example-based specification for the prototypical implementation of the getter method depicted in Figure 4.2. A class declaration template term (i.e. `jtClassDeclaration`) is used to include both the getter method and the field declaration in the specification.

To indicate explicit points of variation, the specification substitutes `?fieldName` for identifier `var` in the implementation. This transforms the source code excerpt in a specification that encompasses multiple pattern instances and captures the relation between each occurrence of the same meta-variable[9].

The code excerpt in the term exemplifies the prototypical implementation of the essential characteristics of the getter method. The detection mechanism of our approach realizes the *example-based semantics* of the template term. We will introduce the semantics in Section 4.3.2. For now, it suffices to mention that several example-based interpretations are considered for each source code excerpt. One interpretation, for instance, identifies getter methods that match the prototypical implementation exactly. Another interpretation recognizes getter methods that have additional instructions. Under the latter interpretation, method `getAge()` in Figure 4.8 will be recognized.

4.3.2 Motivation for the Example-Based Specification Cornerstone

We conjecture that developers tend to think of patterns in terms of examples: code fragments that exemplify their essential characteristics. Example-based specifications are highly descriptive (criterion **CSL2**). They ease the hurdle that unfamiliar specification languages pose to a developer. Consider the example-based specification for the getter method in Figure 4.7. It is arguably more descriptive and accessible than its logic meta programming equivalent in Figure 4.3.

The occurrences of concrete syntax in the state of the art primarily serve to express the *syntactic characteristics* of a *single* program element. Figure 3.10 depicts the CONDATE [Vol06a] specification for the reading of a closed file. Concrete syntax is used to specify the syntax of the instructions that open, read and close the file `%F` (e.g. `close(%F)`). The data flow characteristics are expressed through the occurrences of `%F`. The control flow characteristics of the pattern (i.e. the sequencing of the instructions) cannot be expressed in concrete syntax.

Example-based specifications, in contrast, adopt whole *source code excerpts* of a coarse granularity. The concrete syntax of complete method declarations and class declarations can be used as example-based specification. Source code excerpts serve to exemplify the prototypical implementation of the *syntactic, structural, data flow and control flow characteristics* of a pattern (criterion **CSL1**). They are interpreted as such by the detection mechanism which realizes their example-based semantics.

This way, developers do not have to quantify over reified program information to express the characteristics that are exemplified by the source code excerpt. The details of the program information *and* its reification are hidden (criterion **CSL5**). The quantification-related shortcomings of the logic meta programming cornerstone (cf. Section 4.2.2) are overcome. Depending on which one is more convenient, either a logic-based or an example-based specification can be used. Combinations of both are supported as well.

[9] To account for getter methods that have parameters, the specification uses a naming convention for meta-variables. A meta-variable that ends with a `List` suffix stands for a collection of concrete syntax productions. It will be bound to a collection of abstract syntax tree nodes upon resolution of the template term it resides in. Meta-variables are dynamically typed. This naming convention is therefore merely a parser directive.

Example-Based Semantics for Source Code Excerpts

The detection mechanism realizes the *example-based semantics* of the source code excerpts in an example-based specification. The same excerpt can exemplify both non-behavioral as well as behavioral characteristics of the prototypical implementation of a pattern.

For one user, the syntactic characteristics of the instructions in the excerpt might exemplify the prototype. For another user, the control and data flow established by the same instructions might exemplify the prototype. The detection mechanism has to account for all possibilities. It should therefore consider several example-based interpretations for a source code excerpt.

In our research artifact, the following interpretations are predefined:

Syntactic interpretation Under this interpretation, the syntactic characteristics of the source code excerpt exemplify the pattern. It is strict in the sense that only perfect matches are reported. Matching program elements do not exhibit any other characteristics than those exemplified by the excerpt. The points of variation among the matches are restricted to those indicated explicitly by the meta-variables in the excerpt (i.e. the explicit points of variation).

The prototypical getter method depicted in Figure 4.2 is a perfect match for the specification in Figure 4.7. None of the getter methods depicted in Figure 4.8 matches the specification under this interpretation. Matching classes have one matching field declaration and one matching method declaration. The field is private and has no other modifiers. The method only differs from the prototype in its name and in the operand of the `return` statement. This operand has to match the field.

The search strategy associated with this interpretation quantifies over reified abstract syntax trees. It uses the amount of declarations (statements) in candidate classes (methods) to direct its search. It is very cost-effective, but supports no implicit points of variation among the matches.

Lexical interpretation This interpretation is a less restrictive version of the syntactic interpretation. Matches have to exhibit the syntactic characteristics exemplified by the excerpt, but are allowed to exhibit additional ones. However, the lexical relations among the elements have to be the same as the ones among the corresponding elements in the excerpt. If a statement in a specification is preceded by a local variable declaration, for instance, matching statements have to be preceded by a matching variable declaration as well.[10]

The prototypical getter method matches the specification under this interpretation. Method `getAge()` from Figure 4.8 is also reported as a match. Matching classes feature a matching field and a matching method. The field and method must be declared in the lexical scope of the class declaration. The field is allowed to have additional modifiers. Likewise, the method lexically features a `return` statement of which the operand matches the field. Additional statements are allowed. The `return` statement is also allowed to be nested within other statements.

The search strategy associated with this interpretation is costlier than the previous one. It accounts for implicit points of variation among the instances of

[10] For performance reasons, the implementation imposes this ordering constraint on all statements. Matches for `?x` *lexically* (i.e. based on line numbers) precede matches for `?y` in the specification `{?x; ?y;}`.

a pattern (i.e. different implementations of the same characteristic) as required by criterion **CDM3**.

Control flow interpretation Under this interpretation, the control flow characteristics of the source code excerpt exemplify the pattern. Consider the specification `jtStatement(?block){ {?x; ?y;} }` for a block with two instructions. Matches for *?x* are evaluated, in the control flow of the block, before the matches for *?y* during at least one execution of the program. Other instructions are allowed between these matches.

The control flow established by the getter method is very basic. Matching methods have an expression in their control flow that matches the field. This expression is followed in the flow by a `return` statement. The operand of this statement matches the expression.

The search strategy associated with this interpretation examines control flow graphs. Implicit points of variation are supported:

- non-specified instructions are allowed in the graph
- the strategy only requires that the `return` statement is reachable through at least one path (i.e. existentially qualified). It does not require that all paths through the graph feature the `return` statement (i.e. universally qualified). The latter is more expensive.
- the strategy crosses method boundaries in the search for matching instructions (i.e. it is inter-procedural). However, statements are matched intra-procedurally. Otherwise, methods invoking a getter would for instance be recognized as a getter method themselves.

The specification in Figure 4.7 exemplifies the data flow characteristics of the getter method through the *?fieldName* meta-variable. It substitutes for the name of the field declaration as well as the operand of the `return` statement. Both are required to unify.

In isolation from the other cornerstones of our approach, the example-based interpretations will incorrectly report method `notGettingAge(Integer)` in Figure 4.8 as an instance of the getter method.[11] At the same time, getter method `retrieveBirthDay()`, will not be recognized. The *domain-specific unification* cornerstone will remedy these shortcomings by incorporating data flow information in the unification of the *?fieldName* occurrences.

4.3.3 Concrete Instantiation in Brief

Chapter 7 discusses the example-based specification cornerstone in detail, including its instantiation in the artifact used to validate our approach. Here, we briefly clarify how template terms are resolved by the logic meta programming instantiation in which they are embedded. The architectural overview in Figure 4.1 illustrates the relation between both.

At compile-time, the SOUL evaluator parses the source code excerpt of a template term using a definite clause grammar [PW80] (cf. Section A.1) for

[11] Although method `notGettingAge(Integer)` matches the template, it is not a getter method for the field `age` as the definition of the field is shadowed by the parameter of the method. The SOUL version of the specification in Figure 4.3 has the same problem.

```
1  class Person {
2    private Date birthday;
3    private Integer age;
4    private boolean ageDirty;

5    public Date retrieveBirthday() {
6      Logger.log("Birthday retrieved");
7      return this.birthday;
8    }

9    public Integer getAge() {
10     if(ageDirty) age = Calender.now().yearsSince(retrieveBirthday());
11     return age;
12   }

13   public Integer notGettingAge(Integer age) {
14      return age;
15   }

16   public Date wastefulGetBirthday(int cyclesToWaste) {
17     Date val = (Date) indirectReturn(birthday, cyclesToWaste);
18     return val;
19   }

20   public Object indirectReturn(Object o, int delay) {
21     if(delay == 0)
22       return o;
23     else
24       return indirectReturn(o, delay - 1);
25   }
26 }
```

Figure 4.8: Implementation variants of the getter and setter method best practice patterns in Java.

Java [GJSB00].[12] The predicate name of the template term is used as the starting rule for the grammar. This results in a forest of ASTs for the excerpt.[13]

Each example-based interpretation transforms each AST for the source code excerpt into a logic query. This results in a set of logic queries. The predicates in these queries stem from the CAVA library (cf. Section 5.2) and quantify over the reified program representation. At run-time, template terms are resolved by backtracking over each generated query.

4.4 Cornerstone: Domain-Specific Unification

This section introduces the domain-specific unification cornerstone of our approach. It is detailed in Chapter 6.

Unification is the process of computing a substitution θ, a function from variables to terms, that unifies two terms s and t such that $s\theta = t\theta$. It is an

[12] Template terms only support the concrete syntax of Java 1.4 for pragmatic reasons. However, the program representation supports Java 1.5.

[13] The combination of concrete syntax and untyped meta-variables in the excerpt is often ambiguous. Template terms support this inherent ambiguity by transparently considering all possible abstract syntax trees. This way, the user is not burdened with the disambiguation and is assured that the intended specification is always considered. However, parser directives for disambiguation are available.

essential ingredient of the proof procedure in logic programming (i.e. resolution [Rob65]). Unification has applications outside automated theorem proving as well (cf. Knight [Kni89] for a survey). In pattern detection, *s* stems from the specification and *t* from the reified program representation (which may include variables).

The detection mechanism of our approach employs a *domain-specific unification* procedure which differs from the general-purpose unification procedure. Its comparisons of program representation elements are specifically tailored to the pattern detection domain.

Pattern specifications express such comparisons to relate program elements: either explicitly by equating variables (last condition in Figure 4.3) or implicitly through multiple occurrences of a variable (`?fieldName` in Figure 4.7).

Concretely, the domain-specific unification procedure differs from the general-purpose procedure on the following points:

- The domain-specific unification procedure treats reified program elements different from other terms. The comparison of two reified program elements can succeed where the general-purpose procedure fails. Through domain-specific comparisons, different implementations of the same characteristic are treated uniformly. As a result, our detection mechanism supports *implicit points of variation* among the instances of a pattern (criterion **CDM3**).

- To recognize implicit variation points, the procedure incorporates the results of *whole-program*[14] data flow analyses (cf. Section 2.5.4) in domain-specific comparisons of *individual* program elements.

  A *semantic analysis* [ASU86] ensures correctness. When a fully qualified and an unqualified type are compared, for instance, the import declarations of their compilation units are taken into account. A *points-to analysis* [Hin01] enhances identification efficacy. When two expressions are compared, syntactic deviations are allowed as long as they may alias at run-time.

  Users benefit from the results of these analyses without being exposed to their intricate details (criterion **CSL5**). Concretely, multiple occurrences of a meta-variable express *data flow characteristics* (in positions where they stand for expressions).

- *Reified program elements* (cf. Section 4.2) unify with *compound terms*, even if the reified version of the element is not a compound term. We will show that this ensures the descriptiveness of logic rules (criterion **CSL2**) by enabling the natural use of unification to quantify over reified program information.

By adopting example-based specifications, our approach remedies the quantification-related shortcomings of logic meta programming. The unification-related shortcomings of its founding cornerstone will be remedied by adopting domain-specific unifications. We will identify these shortcomings in Section 4.4.2.

4.4.1 Running Example Revisited

Example-based and logic meta programming specifications share the same unification procedure. Clearly, variable bindings need to be consistent across the conventional terms and template terms in a specification.

[14] A whole-program analysis takes the code of the whole program into account.

Here, we illustrate domain-specific unification on the example-based specification for the getter method. We will revisit the logic rule for the getter method after our discussion of its unification-related shortcomings.

Example-Based Specification for the Getter Method Revisited

The unification procedure *complements the example-based interpretations* of the specification in Figure 4.7. The matches for the specification differ under each interpretation. Under the syntactic interpretation, for instance, matching methods consists of a single matching `return` statement. Under the lexical interpretation, in contrast, matching methods feature this statement lexically. However, the statement must return the value of the field under each interpretation. The specification exemplifies this data flow characteristic through the occurrences of the *?fieldName* variable. It substitutes for the name of the field as well as the operand of the `return` statement. The reifications of both are required to unify.

Assume for a moment that both are reified as string constants. Using the general-purpose unification procedure, method `notGettingAge(Integer)` from Figure 4.8 would be reported as a match for the getter method under all interpretations. Each occurrence of the *?fieldName* variable is bound to the same string. However, the method is not a getter method. The definition of the field is shadowed by the parameter of the method. Getter method `retrieveBirthday()`, on the other hand, would not be recognized under any interpretation. The reification of its operand (`'this.birthday'`) does not unify with the reification of the field (`'birthday'`).

We could avoid this problem by using a reification that makes the "references" relation between the operand and the field explicit. However, other patterns might require different reifications. Moreover, the bindings for the *?fieldName* variable would expose users to the details of the reification.

Complemented by the domain-specific unification procedure, all example-based interpretations treat both methods correctly. To determine whether the reified operand (an `org.eclipse.jdt.core.dom.SimpleName` instance) and reified field (an `org.eclipse.jdt.core.dom.VariableDeclarationFragment` instance) unify, the results of the *semantic analysis* are consulted first. Unification succeeds if the operand definitely references the field according to the semantic analysis. This can be determined for certain variable reference expressions (i.e. a field access or a reference to a parameter), but not for all expressions in general. For those expressions, the results of the alias analyses are consulted (i.e. the inter-procedural points-to analysis and the intra-procedural must-alias analysis). As the intra-procedural must-alias analysis only offers aliasing information about two expressions within a method, the results of the inter-procedural points-to analysis are consulted for this example. These determine whether the operand and the field may-alias at run-time (i.e. when their respective points-to sets have a non-empty intersection). Matches for the specification therefore include methods that return the value of the field without referring to it directly. For instance, a getter for a field that aliases with the field in the specification. Method `wastefulGetBirthday(int)` is another example.

Unification based on points-to analysis supports more implicit points of variation among pattern instances than unification based on semantic analysis or must-alias analysis. However, the higher recall comes at the cost of false positives caused by the imprecision of the points-to analysis (cf. Section 2.5.4). Matches for the spec-

```
1 if compilationUnit(packageDeclaration(simpleName(['examples'])),?,?types) isCompilationUnit,
2    ?types contains: ?type

3 if ?compilationUnit isCompilationUnit,
4    ?compilationUnit hasPackage: ?package,
5    ?package hasName: ?name,
6    ?name isSimpleName,
7    ?name hasIdentifier: ['examples'],
8    ?compilationUnit compilationUnitHasTypes: ?types,
9    ?types contains: ?type
```

Figure 4.9: SOUL queries quantifying over all types defined by compilation units in the package named `examples`.

ification also include `indirectReturn(Object,int)` as a getter method for the `birthday` field. This method returns the value of the field for at least one of its invocations.

Clearly, users should be able to discern matches identified by the points-to analysis from those identified by the semantic analysis and the must-alias analysis (cf. criterion **CDM2**). The *fuzzy logic* cornerstone of our approach will provide a ranking for matches.

4.4.2 Motivation for the Domain-Specific Unification Cornerstone

The detection mechanism of our approach uses the same domain-specific unification procedure for example-based and logic meta programming specifications. The general-purpose unification procedure of logic meta programming would result in operational and convoluted specifications.

We identify the unification-related shortcomings of the logic meta programming cornerstone. We revisit the logic rule for the getter method using the domain-specific unification procedure which remedies these shortcomings.

Unification-Related Shortcomings of LMP

The example-based specification for the getter method demonstrated the need for domain-specific comparisons of program elements. Without these comparisons, alternative implementations of the pattern's data flow characteristics would not be recognized. Worse, a naive comparison of variable references and declarations would lead to false positives being reported. We illustrated these issues using a hypothetical reification to strings.

In general, logic meta programming tools employ a reification to compound terms. Such a reification enables the natural use of unification to quantify over program elements in a descriptive manner. Candidate elements can be required to unify with a compound term that does not stem from the program representation, but unifies with the desired program elements. The first query in Figure 4.9 uses this technique to quantify over all types in compilation units that are declared in the package named `examples`.[15]

[15] Upon backtracking over the first condition, the predicate `isCompilationUnit/1` binds its argument to a reified compilation unit which has to unify with a partially ground compound term. This succeeds only for the compilation units in the package named `examples`. When successful, `?types` is bound to the types in the compilation unit.

Example-based and logic specifications share the need for comparisons that are tailored to pattern detection. In logic meta programming, however, the unification procedure is hard-wired. Users have to implement domain-specific comparisons themselves.

The reification determines which comparisons of program elements are performed by the general-purpose unification procedure when their reified versions are unified. A reification that maps structurally equivalent program elements to the same compound term, for instance, gives rise to structure-based comparisons in the general-purpose unification procedure. The first query in Figure 4.9 requires such a reification. In particular, a reification that reifies the children of an AST node as the arguments of a corresponding compound term.

To support other domain-specific comparisons, additional reifications of the same program element have to be introduced. A reification in which modifier lists are sorted, for instance, gives rise to domain-specific comparisons that consider `public static` and `static public` declarations equivalent. A reification of the form `astNode(uniqueidentifier)`, on the other hand, gives rise to identity-based comparisons of AST nodes using the general-purpose unification procedure.

Users have to implement domain-specific comparisons by quantifying over the most appropriate reification manually. This results in operational and convoluted queries that are not descriptive. The identity-based reification, for instance, does not give rise to structure-based comparisons in the general-purpose unification procedure. In the second query of Figure 4.9, structure-based comparisons are therefore implemented manually. It is less descriptive and more convoluted than the first query.

The domain-specific unification cornerstone remedies the above shortcomings by treating reified program elements different from other terms. Domain-specific comparisons are used in their unification to support implicit points of variation among pattern instances (e.g. to recognize different implementations of a data flow characteristic as described in Section 4.4.1). This precludes users from implementing such comparisons manually which leads to operational queries.

As multiple reifications are no longer necessary, users are only exposed to the details of one reification. Regardless of this reification, the domain-specific unification procedure unifies reified program elements with compound terms that are structurally equivalent. This facilitates the natural use of unification to quantify over reified AST nodes as illustrated by the first query in Figure 4.9.

LMP Specification for the Getter Method Revisited

SOUL employs an identity-based reification which forgoes a transcription to compound terms. The reified version of an AST node is the AST node itself (i.e. an object). As a result, the general-purpose unification procedure only unifies identical objects.[16] The operational nature of the logic rule for the getter method in Figure 4.3 is due to the combination of a compound-free reification with a general-purpose unification procedure. Its reporting of the false positive `notGettingAge(Integer)` and its failure to recognize getter method

[16]The procedure tests for the result of the Smalltalk equality message, =, sent to one of the objects. Smalltalk proxies for Java objects respond to this message with an invocation of the Java `equals(Object)` method which is implemented as object identity == for the `org.eclipse.jdt.core.dom.ASTNode` instances in the program representation.

```
1  if ?method isMethodDeclaration,
2     ?fieldDeclaration fieldDeclarationHasFragments: ?fragments,
3     [?fieldDeclaration parentTypeDeclaration] equals: [?method parentTypeDeclaration],
4     ?fragments contains: variableDeclarationFragment(simpleName(?identifier), ?, ?),
5     ?method methodDeclarationHasBody: block(nodeList(<returnStatement(simpleName(?identifier))>))

6  if ?method isMethodDeclaration,
7     ?fieldDeclaration fieldDeclarationHasFragments: ?fragments,
8     [?fieldDeclaration parentTypeDeclaration] equals: [?method parentTypeDeclaration],
9     ?fragments contains: variableDeclarationFragment(?name, ?, ?),
10    ?method methodDeclarationHasBody: block(nodeList(<returnStatement(?name)>))
```

Figure 4.10: SOUL queries illustrating domain-specific unification in the detection of the getter method.

`retrieveBirthDay()` (see Figure 4.8) are due to to a naive, manual implementation of the domain-specific comparison of variable references and declarations.

The two logic queries depicted in Figure 4.10 are alternative specifications for the getter method. Both are more succinct and descriptive than the logic rule depicted in Figure 4.3. They rely on the domain-specific unification cornerstone to overcome the unification-related shortcomings of logic meta programming.

The first query relies on the unification of reified program elements (i.e. `org.eclipse.jdt.core.dom.ASTNode` instances) with compound terms. The query detects the same instances as the convoluted logic rule depicted in Figure 4.3, including the false positive `notGettingAge(Integer)`. The query differs from the rule in its selection of the variable declaration fragment among the *?fragments* in the field declaration (i.e. line 4 in the query versus line 8 in the rule). In the query, the second argument to the `contains:/2` predicate is a compound term that will unify with each reified fragment upon backtracking. The arguments of the term correspond to the name, extra dimensions and initializer children of the AST node for the variable declaration fragment.[17] An anonymous variable in the last two argument positions indicates that we are not interested in their values. The first argument of the term is another compound term. It will unify with the name child of the node. As a result, the *?identifier* variable gets bound to the string that identifies the fragment. Line 4 of the query has the same effect as lines 8–10 in the rule.

The condition on line 5 relies on the unification of `ASTNode$NodeList` instances (i.e. collections) with a single-argument compound term containing a reification of their contents as a logic list.[18] The condition states that the block in the body of the method declaration is made up of a single statement: a return statement of which the operand is a simple name named after the *?identifier* of the field. This condition has the same effect as the 7 conditions on lines 11–17 of the original logic rule.

The `notGettingAge(Integer)` false positive is not reported by the second query. The query relies on domain-specific comparisons of the binding for the *?name* variable on line 9 (the name child of the variable declaration fragment) and the binding for the *?name* on line 10 (the operand of the return statement). Once the restriction on the amount of statements in the method is lifted, this query cor-

[17] Their concrete syntax counterparts are produced by the grammar rule `VariableDeclarationFragment ::= Identifier { [] } [ = Expression ]` in EBNF notation.

[18] The SOUL list `<1,2|?tail>` is equivalent to the Prolog list `[1,2|Tail]`.

rectly identifies all getter methods in Figure 4.8. In fact, as domain-specific comparisons are also implemented for variable declaration fragments, there is no need to extract the name child from the fragment on line 9.

These queries illustrate how the domain-specific unification cornerstone ensures the descriptiveness of logic-based pattern specifications.

4.4.3 Concrete Instantiation in Brief

Chapter 6 discusses the domain-specific unification cornerstone in detail, including an enumeration of the domain-specific comparisons implemented in our research artifact.

The architectural overview in Figure 4.1 illustrates the relation between this cornerstone and the program representation. Domain-specific comparisons are defined on reified program elements (i.e. `org.eclipse.jdt.core.dom.ASTNode` instances) using the double dispatching idiom. To support implicit points of variation, these comparisons use the parts of the program representation that are not reified: the results of the semantic analysis, the must-alias analysis and the points-to analysis (cf. Section 4.2.3).

It is important to note that unifying a compound term with a reified program element is meant as convenient syntactic sugar (i.e. for a condition that restricts the type of the node and subsequent unification conditions over its child nodes). Our implementation precludes other uses. It distinguishes regular compound terms (uninstantiated compound terms) from compound terms that have been unified with a reified program element (instantiated compound terms). The latter represent the object they are instantiated to. Unifying two instantiated compound terms therefore amounts to unifying the objects they are instantiated to.

Likewise, within Smalltalk terms (cf. Section 4.2.1), a logic variable bound to an instantiated compound term evaluates to the object the term is instantiated to. An uninstantiated compound term, in contrast, evaluates to the Smalltalk implementation of the compound term itself (i.e. a `Soul.CompoundTerm` instance). The following predicate can be used to discern uninstantiated compound terms from objects and instantiated compound terms:

```
1  +?x isInstantiatedTo: [?x]  if
2     [(?x isKindOf: Soul.CompoundTerm) not]
```

4.5 Cornerstone: Fuzzy Logic

The fourth cornerstone of our approach consists in using fuzzy logic rather than the two-valued logics conventionally employed in pattern detection. It provides our detection mechanism a theoretical foundation to rank the results it reports. This ranking facilitates assessing a large amount of results. The fuzzy logic cornerstone can be incorporated in any tool based on a logic formalism with a machine-executable proof procedure.

Incorporated in a LMP tool, this cornerstone gives rise to a fuzzy logic programming language. Our fuzzy variant of SOUL is representative for the many "fuzzy Prolog" systems that exist, but more advanced instantiations have been devised. However, it constitutes an otherwise rare application of fuzzy logic in software pattern detection (fuzzy graph rewrite rules are used by the fuzzy FUJABA variant discussed in Section 3.2.3). We detail the fuzzy variant of SOUL in Section 6.1.

Fuzzy Logic Programming

Analogous to the partial membership degrees of a fuzzy set [Zad65], fuzzy logics (cf. Petr Hájek and Godo [Háj98] for an introduction) assign a degree of truth to logic propositions. One proposition may be absolutely true, while another may be true to an extent between absolute truth and absolute falsity. Fuzzy logics are logics of quantified truth. In sharp contrast to probabilistic logics, fuzzy logics are truth-functional: the truth of a formula is only determined by the truth of its constituents. As different semantics can be given to the logical connectives $\wedge$, $\vee$ and $\neg$, there exist many different kinds of deductive fuzzy logic. As a notion of proof, these logics use modus ponens.

In fuzzy logic programming, a resolution procedure is used as machine-executable proof procedure. Lee [Lee72] was the first to extend the classical resolution procedure to handle partial truths, initiating a plethora of "fuzzy Prolog" systems. These all differ in the way they model the logical connectives as well as in whether or not they allow fuzzy facts, fuzzy rules or fuzzy constants. A detailed historical overview of the resulting programming languages can be found in Alsinet's dissertation [Als01].

In our fuzzy variant of SOUL, logic facts and rules can be annotated with truth degrees. The language is close to F-PROLOG [LL90] in that its resolution procedure quantifies truth in a similar manner. In addition, our unification procedure quantifies unification. However, we support only real-valued partial truth values in the interval $]0,1]$, while the F-PROLOG system supports fuzzy numbers.

Fuzzy logic rules are of the following form. Like standard rules, they have a head q and a body $q_1,\ldots,q_n$. In contrast to standard rules, they are annotated by a partial truth degree $c \in]0,1]$. This degree is interpreted as the confidence the programmer has in the conclusion q reached by the rule given the absolute truth of the subgoals in its body.

$$q : c \textit{ if } q_1,\ldots,q_n.$$

Developers can use these annotations to establish a *ranking among alternative logic meta programming specifications* for a pattern. We use this feature internally to establish a *ranking among the example-based interpretations* of a template term. Recall from Section 4.3.3 that a template term is translated to a different logic query by each example-based interpretation.

Fuzzified Resolution Procedure

The essential difference between the fuzzified resolution procedure and the standard procedure lies in the quantification of the deduced answer sets for a goal. Our fuzzy logic programming language computes the truth degree of the conclusion q as the product of c and the minimum of the truth degrees of the subgoals $q_1 \ldots q_n$. Hence, we interpret conjunction and implication as minimum and product respectively. Negation is interpreted as complement.

Fuzzified Unification Procedure

An analogously fuzzified unification procedure quantifies the extent to which two terms unify. Unifying two terms results either in failure, or in a unification degree. The resolution procedure incorporates unification degrees in its quantification of

```
1  15 isAmountSoldOf: flowers.
2  (chips hasAttractivePackaging) : [9/10].
3  (chips isWellAdvertised) : [6/10].

4  ?product isPopular if
5    ?amount isAmountSoldOf: ?product,
6    [?amount > 10].

7  (?product isPopular) : [8/10] if
8    ?product hasAttractivePackaging,
9    ?product isWellAdvertised.
```

Figure 4.11: A fuzzy SOUL program illustrating quantified resolution.

the answer sets for a goal. This way, unification degrees are propagated to the results.

Through the propagated degrees, the domain-specific unification cornerstone communicates which program analyses were used in the unification of reified program elements. For instance, whether the unification of two expressions required the results of the points-to analysis (i.e. they may alias in an execution of the program) or the results of the must-alias analysis (i.e. they definitely alias in every execution of the program). This allows us to establish a ranking among *the results reported by a single example-based interpretation* of a template term and the solutions to a logic query.

An Illustrative Fuzzy Logic Program

Consider, as an introductory example, the fuzzy logic program depicted in Figure 4.11. It models the vague concept of the popularity of a grocery item. The standard logic rule on lines 4–6 states that any product of which more than 10 items have been sold is definitely popular. Well-advertised products with an attractive packaging are considered popular by the fuzzy logic rule on lines 7–9. The latter has an associated truth degree of `[8/10]`, an instance of the Smalltalk `Fraction` class. This expresses our confidence in the conclusion of the rule given the absolute truth of the goals in its body. The parentheses in the head of the rule are not required and are only added for clarity. The background information of the program states that 15 `flowers` have been sold, while the product `chips` has a fairly attractive packaging and has been advertised reasonably well.

From the depicted program, we can derive that the product `chips` must be fairly popular. It is found as a solution to the query `if ?product isPopular` with a reasonably large partial truth degree of $\min(\frac{9}{10}, \frac{6}{10}) \cdot \frac{8}{10} = \frac{48}{100}$. The product `flowers`, on the other hand, is definitely popular.

4.5.1 Running Example Revisited

Figure 4.12 depicts the quantified results for an example-based specification of the getter method. The specification itself is shown in the bottom-left window of the figure. It is a logic query that consists of a single condition which is a template term annotated by the variable *?degree*. This variable will be bound to the partial truth degree that is computed for the term it annotates. The solutions for the query

are shown in the left column of the upper left window. They consist of bindings for the *?class*, *?field* and *?methodName* variables (bindings for the other variables are not shown). All solutions are getter methods from the `Person` class depicted in Figure 4.8. Note that the false positive `notGettingAge(Integer)` is not reported.

Figure 4.12 illustrates the reification of the founding logic meta programming cornerstone in addition to the fuzzy logic, example-based specification and domain-specific unification cornerstones of our approach. The topmost inspector window on the right was obtained by inspecting the binding for the *?field* variable in the first solution. This is an instance of `org.eclipse.jdt.core.dom.SimpleName`. In the evaluation pane of the window, we sent it the message `getParent` and printed the result. The parent of this instance in the AST is the variable declaration fragment from the field declaration. Next, we sent it the message `pointsTo` and inspected the results. The inspector window in the bottom shows the points-to set for the variable declaration fragment as retrieved from the points-to analysis results.

The right column of the results window lists the truth degree associated with each solution. These degrees establish a ranking among the solutions. Solutions with a higher degree are less likely to be false positives. A single result can be identified by multiple example-based interpretations of a template term. For each result, Figure 4.12 lists only the largest of its associated truth degrees. All methods match the term under the control flow interpretation, for instance, but never with a truth degree higher than 8/10.

Ranking among matches from different example-based interpretations

Truth degrees for solutions identified under the syntactic, lexical and control flow interpretation of a template term (cf. Section 4.3.2) can be no larger than 1, 9/10 and 8/10 respectively. This ranking reflects the projected similarity of solutions to the source code in the term. Under the syntactic interpretation, only perfect matches for a template term are identified. None of the getter methods in Figure 4.8 are perfect matches for the template term. Under the lexical interpretation, matches can deviate from the source code in the template term. In addition to the specified return statement, methods `getAge()` and `retrieveBirthDay()` feature a logging and a lazy initialisation instruction respectively. They are identified under the lexical interpretation.

Ranking among matches from a single example-based interpretation

Methods `getAge()` and `retrieveBirthDay()` match the template term under the lexical interpretation, but have an associated truth degree of 0.81 which is smaller than the maximum of 9/10 for this interpretation.

Truth degrees can vary among the matches identified by a single example-based interpretation of a template term. The more characteristics a match exhibits in addition to the ones that are exemplified by the source code excerpt in the term, the smaller its associated truth degree. Under the lexical interpretation, for instance, a method with 3 instructions is considered a better match for a specification with 1 instruction than a method with 6 instructions.

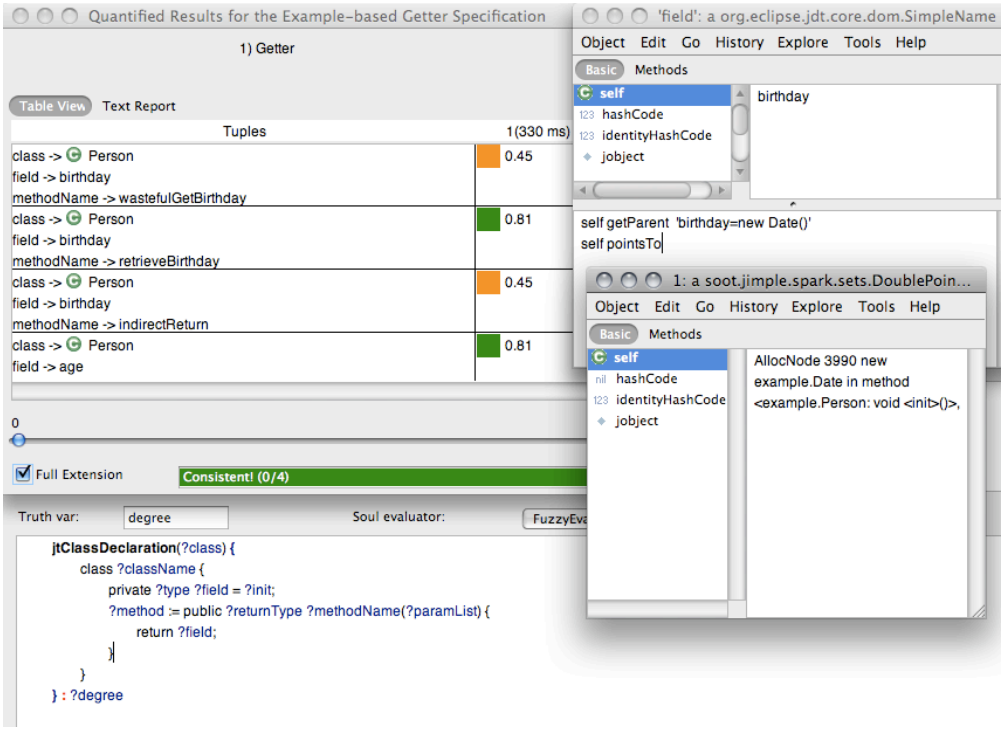


Figure 4.12: Quantified results for the example-based specification of the getter method in Figure 4.7 matched against the implementations in Figure 4.8.

Influence of domain-specific unification degrees

As discussed in Section 4.4.1, the domain-specific unification procedure complements the example-based interpretations of a template term. All methods are identified because the reified operand of their return statement unifies with the reified field.

Unification degrees lower the truth degrees computed for a match. Unification succeeds either because the operand is a variable reference to the field (according to the semantic analysis) or because the operand and the field may-alias at runtime (according to the points-to analysis). The former is the case for the aforementioned methods. The latter is the case for methods `wastefulGetBirthday()` and `indirectReturn(Object,int)`. Because the operand and field do not necessarily alias in all invocations of these methods, the unification degree halved the truth degree computed for the lexical interpretation of the template term. As a result, the false positive `indirectReturn(Object,int)` can be discerned more easily among all results.

4.5.2 Motivation for the Fuzzy Logic Cornerstone

The fuzzy logic cornerstone provides our detection mechanism with a *theoretical foundation to rank its results*. This ranking facilitates assessing a large amount of results (criterion **CDM2**). A truth degree is associated with each result. Analogous to partial set memberships, the reported degrees quantify the extent to which each

result can be considered an instance of the specified pattern. In other words, the extent to which reported instances exhibit the characteristics in a specification is quantified. The smaller this extent, the more likely the reported instance is a false positive.

Communicating this likelihood is desirable when the search strategies employed by the detection mechanism implement different trade-offs with respect to cost, precision and recall. Because of domain-specific unification, this is the case for logic queries with and without template conditions. Unification based on points-to analysis supports more implicit variation points than unification based on semantic analysis. This higher recall comes at the cost of false positives caused by the imprecision of the points-to analysis. Their respective unification degrees reflect this trade-off.

The fuzzy resolution procedure combines the truth degrees of all conditions in a query and takes unification degrees into account.

Concrete Ranking of Results

The logic rules used in the resolution of a logic term and the example-based interpretation used in the resolution of a template term establish an upper bound for the truth values of their solutions:

- Figure 4.13 illustrates this for logic terms. It depicts a fuzzy query and two fuzzy rules. The second, recursive rule is annotated with a lower truth degree. In the base program, `MPExtender` extends `MPOverrider` which extends `AbstractBaseClass` which extends `Object`. Solutions to the query consists of ancestors of `MPExtender`. In the solutions, the maximum truth degree 1 is associated with the immediate super class `MPOverrider`. It is identified by the first rule. Indirect super classes have a lower associated truth degree that corresponds to the amount of recursive invocations of the second rule. The minimum truth degree of the goals in its body is multiplied with the truth degree its head is annotated with. For `AbstractBaseClass`, this results in a truth degree of $(\frac{999}{1000})^2$.
- The source code excerpt of a template term exemplifies the prototypical implementation of a pattern's essential characteristics. Truth degrees for solutions identified under the syntactic, lexical and control flow interpretation of a template term (cf. Section 4.3.2) can be no larger than 1, 9/10 and 8/10 respectively. This ranking reflects the projected similarity of the solutions to the prototypical implementation of the pattern.

The specific properties of a solution for a term further refine (i.e. lower) the truth degrees computed from the rules and example-based interpretation used in its resolution:

- Solutions that required a unification based on points-to analysis are ranked lower. For both logic terms and template terms, the upper bound established by the logic rules and example-based interpretation is lowered by unification degrees. The unification degree associated with domain-specific unification based on points-to analysis, for instance, halves the truth degree computed by a resolution without unification degrees. In general, truth degrees for a goal are computed by multiplying the unification degree with the truth value computed by the resolution without unification degrees.

```
1  +?classDeclaration extends: ?superclassTypeDeclaration if
2    ?classDeclaration equals: classDeclaration(?,?,?,?,?superclassType,?,?),
3    ?superclassTypeDeclaration typeDeclarationForType: ?superclassType

4  (+?classDeclaration extends: ?ancestorDeclaration) : [999/1000] if
5    ?classDeclaration equals: classDeclaration(?,?,?,?,?superclassType,?,?),
6    ?superclassTypeDeclaration typeDeclarationForType: ?superclassType,
7    ?superclassTypeDeclaration extends: ?ancestorDeclaration
```

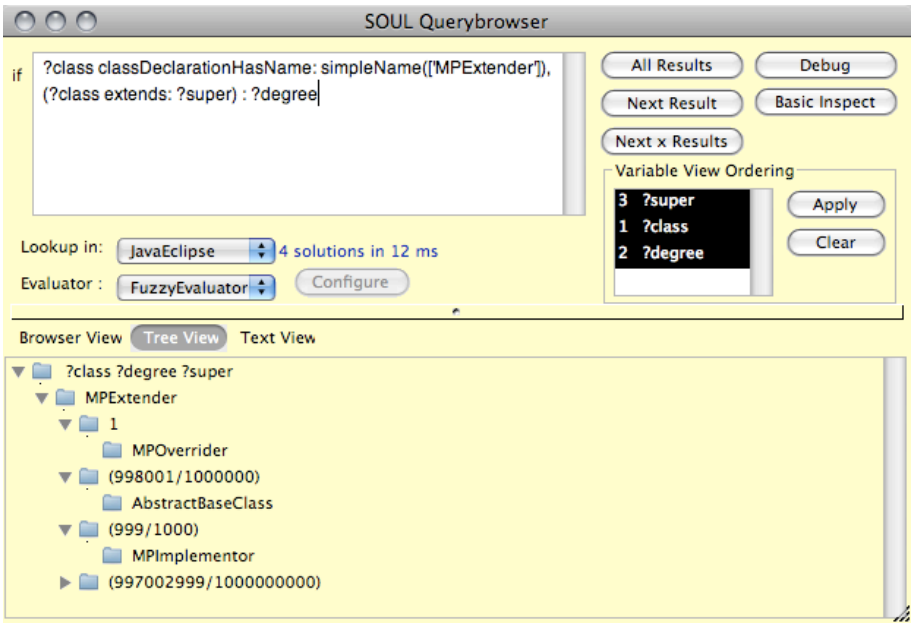


Figure 4.13: Quantified results for a fuzzy SOUL query and the fuzzy rule defining the predicate used in the query.

- For template terms, truth degrees differ among solutions identified under a single example-based interpretation. The more characteristics a solution exhibits in addition to the ones that are exemplified, the smaller its associated truth degree. For a method specification that only exemplifies a `public` modifier, a method with a single modifier is a better match than a `public static` method. For a class specification that exemplifies 3 members, a class with 5 members is a better match than a class with 7 members. This is true for all example-based interpretations —even for the control flow interpretation under which only the control flow characteristics of the template exemplify the pattern.

Note that the sole intent of the computed truth degrees is to facilitate user assessment of a large amount of results. The ranking established by these degrees is more important than the degrees themselves. However, fuzzy logic programming offers several ways to further manipulate the degrees that are computed for a goal. We will discuss these in Section 6.1.

4.5.3 Concrete Instantiation in Brief

Unification degrees computed by the domain-specific unification procedure are discussed in Chapter 6. Chapter 7 details the truth degrees computed by resolving the example-based interpretations of a template term.

The fuzzy logic cornerstone is incorporated into SOUL, the logic meta programming instance described in Chapter 5, by specializing the object-oriented implementation of its resolution procedure.

4.6 Cornerstone: Open Implementation

This section introduces the fifth cornerstone, user-extensibility through open implementations, which crosscuts the implementations of the other cornerstones. Each implementation presents a meta-interface through which existing search strategies of the detection mechanism can be altered and user-defined ones can be implemented (criterion **CDM4**). The meta-interface gives clients control over the search strategies without exposing them to all of their implementation details. The abstraction levels of the meta-interfaces differ.

4.6.1 Open Implementation of the Logic Meta Programming Cornerstone

In a sense, the implementation of SOUL [Wuy98, Wuy01, Sou08] (i.e. the instantiation of our founding cornerstone) has been open since its conception. Its hybrid language characteristic allows quantifying over any object that is reachable in the Smalltalk run-time image, including objects from its implementation.

Consider the logic rule for the `isInstantiatedTo:/2` predicate in Section 4.4.3. It illustrates that the Smalltalk implementation of a logic term can be accessed (e.g. a `Soul.CompoundTerm` instance in the rule). By manipulating these objects in a logic rule, the implementation can be changed from within the language itself. Several SOUL rules rely on this functionality. They can access the current environment, current call stack, invoke the unification procedure, create resolution results, create new terms, change lexical addresses etc.

Clearly, defining custom search strategies through this meta-interface requires knowledge about the internals of SOUL that is on par with that of its implementers. In this regard, defining a meta-interpreter that implements a custom search strategy for logic specifications is a better option. Technically, however, both options are available to knowledgeable users.

4.6.2 Open Implementation of the Fuzzy Logic Cornerstone

The implementation of the fuzzy logic cornerstone specializes the object-oriented implementation of SOUL. It is therefore as open as SOUL itself. Technically, its implementation can be changed from within the fuzzy logic programming language, but not without being exposed to its implementation details.

In the definition of an open implementation by Kiczales [Kic96, KPK94], a meta-interface explicitly hides such implementation details of the primary interface: *"the primary interface provides the functionality and the meta-interface allows the client to adjust the implementation strategy decisions that underlie the primary interface"*.

4.6.3 Open Implementation of the Domain-Specific Unification Cornerstone

The acronym SOUL stands for Smalltalk *Open Unification* Language. The implementation of the unification procedure of SOUL is truly open. Whether two logic terms unify is determined by sending the message `unifyWith:inEnv:` to the Smalltalk implementation of one of the terms with the Smalltalk implementation of the other term as argument. The corresponding methods comprise the meta-interface through which the general-purpose unification procedure can be extended. SOUL already employed a modest extension to accommodate the unification of reified Smalltalk objects. Their implementation of `unifyWith:inEnv:` invokes = on the corresponding Smalltalk objects.

The domain-specific unification procedure of our approach defines additional extensions on reified abstract syntax tree nodes. These are tailored to the pattern detection domain and incorporate whole-program analysis results. The openness of the unification procedure allows users to define additional domain-specific extensions.

In addition to the meta-interface provided by SOUL, our implementation provides an API through which the local variable in the JIMPLE three-address representation (Figure 2.4 depicts an example) that corresponds to an expression in an ECLIPSE AST node can be retrieved. Manually establishing such a mapping is difficult (cf. Section 2.5.1). Program analysis results from the SOOT Java Optimization Framework [VRCG+99] can be queried for this local.

4.6.4 Open Implementation of the Example-Based Specification Cornerstone

As discussed in Section 4.3.3, template terms are resolved by backtracking over logic queries that are generated by SOUL at compile-time. Each query corresponds to an example-based interpretation of an AST for the source code excerpt in the term. The translational semantics is specified and implemented as logic rules.

Rules implementing the predicate "`?template underInterpretation: ?interpretation compilesTo: ?query forResult: ?result`" comprise the meta-interface through which users can define additional example-based interpretations.

The first rule in Figure 4.14 defines the translational semantics for a `jtClassDeclaration(?class){class ?name {...}}` template term. The rule takes four arguments. The *?template* argument is bound to the AST of the source code excerpt within the braces of the term. The second argument *?interpretation* is bound to one of the example-based interpretations. Under this interpretation, the template compiles to the third argument. This is a list of logic conditions that quantifies over the reified program representation. They are evaluated at run-time when the template term is resolved. Variables in the list are compile-time variables. The compile-time *?query* variable, for instance, represents the tail of the conditions list. Compile-time variables can also be bound to a representation of a run-time variable in the query. We refer to such variables as quoted variables. The argument of the first condition in the list is a quoted variable. Its binding stems from the fourth argument *?result* of the rule. The SOUL evaluator will first resolve the template term by evaluating the conditions in the list. Next, it will unify the run-time variable corresponding to the quoted *?result* variable with the argument of the `jtClassDeclaration/1` template term.

A recursive descent through the AST of the class declaration template de-

```
?template underInterpretation: ?interpretation compilesTo: <?result isClassDeclaration|?query> forResult: ?result if
  ?template equals: classDeclaration@(?),
  ?template classDeclarationUnderInterpretation: ?interpretation compilesTo: ?query forResult: ?result

statement(return(?e))
  internalStatementUnderInterpretation: ?interpretation
  compilesTo: <?result equals: returnStatement(?baseExpression)|?expressionQuery>
  forResult: ?result if
    not(?interpretation isControlFlowInterpretation),
    not(?e equals: expression(epsilon)),
    ?baseExpression isNewQuotedVariable,
    ?e expressionUnderInterpretation: ?interpretation
       compilesTo: ?expressionQuery
       forResult: ?baseExpression

statement(return(?e))
  internalStatementUnderInterpretation: ?interpretation
  compilesTo: ?query
  forResult: ?result if
    ?interpretation controlFlowInterpretationHasFlowContext: ?sContext andToBeFound: ?,
    ?sContext equals: contextInTemplate(?flow,?nodesToFollow,<?baseExpression|?extendedNodesToFollow>,?nodesToStayAhead)
    not(?e equals: expression(epsilon)),
    ?baseExpression isNewQuotedVariable,
    ?eContext equals: contextInTemplate(?flow,?nodesToFollow,?extendedNodesToFollow,<?result|?nodesToStayAhead>),
    ?expressionInterpretation controlFlowInterpretationHasFlowContext: ?eContext andToBeFound: [true],
    ?e expressionUnderInterpretation: ?expressionInterpretation
       compilesTo: ?expressionQuery
       forResult: ?baseExpression,
    append(?expressionQuery,<?result equals: returnStatement(?baseExpression)>,?query)
```

Figure 4.14: Open implementation of the translational semantics for a template return statement.

termines the conditions in the tail of the condition list. For the template term in Figure 4.7, the two remaining rules in Figure 4.14 are considered eventually. They specify the translational semantics of a `return` statement. The callers of the `internalStatementUnderInterpretation:compilesTo:forResult:` will generate conditions that quantify over the statements in the program representation. The conditions generated by these rules further constrain candidate statements to `return` statements:

- The first rule is applicable under the syntactic and lexical interpretations. Note how it introduces a new quoted variable *?baseExpression* for the operand of the `return` statement. In the generated query, it is bound at runtime by unifying the statement from the program representation with a compound term (i.e. a domain-specific unification). The base program operand is required to unify with the result of the translation of the template operand. The translational semantics does not refer to any program analysis this may require. Their details are hidden by the domain-specific unification procedure.
- The second rule is applicable under the control flow interpretation. It is complicated because it has to specify the contexts of the return statement and its operand in the control flow. It specifies that the operand has to precede the return statement in the control flow *?flow*. The nodes the operand has to follow in the flow, *?nodesToFollow*, are determined by the callers of the rule. The translation of the template operand can further refine these nodes to *?extendedNodesToFollow*. In the flow, the operand has to stay ahead of the return statement: <*?result* | *?nodesToStayAhead*>. The return statement, on the other hand, has to follow the operand and the nodes the operand had to follow: <*?baseExpression* | *?extendedNodesToFollow*>. The nodes it has to stay ahead, *nodesToStayAhead*, are determined by the callers of the rule.

We will further clarify the translational semantics of the example-based interpretations in Chapter 7. The rules depicted in this section illustrate that the implementation language for additional example-based interpretations is close to the pattern specification language. It results in relatively descriptive specifications of the translational semantics for a template term (w.r.t. the Visitor implementations of other LMP tools that feature concrete syntax [LWL+05a, CGM06b]). Users are exposed to details such as quoted variables and control flow contexts, but are shielded from intricate program analysis results by the domain-specific unification procedure. Different abstraction levels are appropriate for different customization tasks.

4.7 Conclusion

In this chapter, we defined our example-driven approach to pattern detection in terms of four cornerstones and their inter-dependencies: logic meta programming, example-based specifications, domain-specific unification, fuzzy logic and open implementations. Section 1.3 provides a more abstract overview of our approach that is structured according to the dimensions in the design of a pattern detection tool (cf. Section 2.4).

Cornerstone: logic meta programming is the founding cornerstone of our approach. It adopts logic formulas for the specification of a pattern. This merely

requires a reification of the program representation to terms in the formalism. Executing a proof procedure establishes whether program elements exhibit the characteristics specified in a formula. The declarative nature and expressiveness of the resulting logic programs facilitates their use as descriptive pattern specifications.

We identified two shortcomings of logic meta programming that lead to operational and convoluted specifications: quantification over and unification of reified program representation elements. An example of such a specification is the one for the getter method depicted in Figure 4.3. These shortcomings are remedied by the *example-based specification* and *domain-specific unification* cornerstones respectively.

Cornerstone: example-based specifications integrates source code excerpts in the *concrete syntax* of the program under investigation within logic formulas. We refer to these extra-logical terms as template terms. An example-based specification of a pattern corresponds to the *prototypical implementation* of its essential machine-verifiable characteristics. Multiple occurrences of logic variables are the primary means to express variation within a template term, but template and logic terms can be connected through logic connectives (e.g. conjunction, disjunction, negation).

The detection mechanism realizes the example-based semantics of a template term. The same code excerpt can exemplify both non-behavioral as well as behavioral characteristics of the prototypical implementation of a pattern. The detection mechanism has to account for all possibilities. Several example-based interpretations are therefore considered for each template term.

Cornerstone: domain-specific unification treats reified program elements different from other terms. Unifying two reified program elements can succeed where the general-purpose unification procedure fails. Example-based and logic meta programming specifications share the same unification procedure. Clearly, variable bindings need to be consistent across the conventional terms and template terms in a specification.

To enable the natural use of unification to quantify over the program representation, a reified program element unifies with a structurally equivalent compound term —even if the reified version of the program element is not a compound term. To recognize *implicit* variation points (i.e. different implementations of the same pattern characteristic), the domain-specific procedure consults *whole-program* data flow analyses when unifying *individual* reified program elements. A semantic analysis [ASU86] ensures correctness. Scoping rules and import declarations are taken into account. A points-to analysis [Hin01] enhances identification efficacy. Syntactically differing expressions unify if their values may alias. Users benefit from the results of these analyses without being exposed to their intricate details.

Cornerstone: fuzzy logic provides our detection mechanism a theoretical foundation to rank the results it reports. Each result is quantified by the extent to which it exhibits the characteristics in a specification. The smaller this extent, the more likely the reported instance is a false positive. This ranking facilitates assessing a large amount of results.

Fuzzy logic is a logic of quantified truth. Concretely, our detection mechanism associates truth degrees with each result it reports. For the logic terms and template terms in a specification, the truth degree of a result is bounded respectively by the example-based interpretation and logic rules used in their resolution.

The properties of the result itself further refine this upper bound. Solutions are ranked lower if they required a domain-specific unification that could introduce false positives (due to imprecision in the program analyses the unification relies on). To this end, unification degrees are associated with each unification. In addition, results for template terms are ranked lower if they exhibit more characteristics than are exemplified by the template.

Cornerstone: open implementation crosscuts the implementations of the other cornerstone. Each cornerstone presents a meta-interface through which existing search strategies of the detection mechanism can be altered and user-defined ones can be implemented. The meta-interface gives clients control over the search strategies without exposing them to all of their implementation details. The meta-interfaces of each cornerstone are in decreasing order of abstraction: logic rules implementing the predicate `underInterpretation:compilesTo:forResult:` (example-based specification), Smalltalk methods implementing `unifyWith:inEnv:` for reified program elements complemented by an API to query whole-program analysis results (domain-specific unification), quantifying over and manipulating implementation objects through the hybrid language characteristic of SOUL (fuzzy logic and logic meta programming cornerstones). Because the latter exposes users to many implementation details, defining a meta-interpreter that implements a custom search strategy is often a better option at this abstraction level.

Having introduced and motivated the cornerstonse summarized above, the chapter outlined their implementations in our research prototype as a suggestion for their concrete instantiation in a pattern detection tool. Figure 4.1 presents an architectural overview. Our example-driven approach fulfills all of the criteria for a general-purpose pattern detection tool identified in Section 2.6. Table 4.1 lists the individual contributions to our approach in terms of the criteria it helps to fulfill.

CHAPTER 5

INSTANTIATING THE LOGIC META PROGRAMMING CORNERSTONE

This chapter discusses the instantiation of the logic meta programming cornerstone in the prototype that we will use to validate our example-driven approach to pattern detection. The overview chapter introduced and motivated this founding cornerstone. Its instantiation consists of the Smalltalk-Prolog hybrid SOUL *and the* CAVA *predicate library. The latter is a technical contribution of this dissertation. We briefly discuss the syntax and semantics of the former and clarify the implementation of key predicates in the latter. We demonstrate the support provided by logic meta programming for each of the previously identified pattern characteristics.*

5.1 The SOUL Logic Meta Programming Language

The *Smalltalk Open Unification Language* (SOUL) [Wuy98, Wuy01, Sou08] is a logic programming language implemented in —and tightly integrated with— Smalltalk [GR83]. SOUL programs consist of both logic conditions and Smalltalk expressions which transparently exchange Smalltalk objects through logic variables. This *linguistic symbiosis* [GWDD06] already renders SOUL interesting by itself. A meta programming task can be implemented in the paradigm that lends itself most fittingly to the task at hand, even resorting to either the imperative or the declarative paradigm for individual sub-tasks.

The ability to embed Smalltalk expressions in logic rules only detracts from the declarative interpretation of a SOUL program if the embedded expressions are not side-effect free. Although sometimes indispensable, embedding Smalltalk expressions with side-effects exposes the operational interpretation of a SOUL program. To a lesser extent, the same is also true for Smalltalk expressions in conditions of a logic rule that depend on each other's outcome through a shared logic variable. Here, the evaluation order of the conditions is made explicit.[1]

[1] For the second condition in the query "`if ?x equals: [1], ?y equals: [?x + 2]`" to bind

All in all, this situation is similar to the input-output predicates of Prolog. Used wisely, SOUL's hybrid language characteristic will not detract from its declarative nature. We therefore consider SOUL as good an instantiation of logic meta programming as approaches centered around Prolog. After all, SOUL programs can still opt to forgo Smalltalk completely.

5.1.1 Syntax and Semantics in a Nutshell

We briefly introduce the syntax and semantics of SOUL in an informal manner. We restrict our discourse to the features that are necessary to understand the code snippets in this dissertation. For a less dense introduction, we refer the reader to the SOUL website [Sou08]. We stress that the version of SOUL described here differs from the one in the earliest papers.

Smalltalk Terms

For predicates, the Smalltalk keyword syntax was adapted to accommodate linguistic symbiosis [DGJ04]. For compound terms, we will retain the traditional notation (i.e. a functor symbol followed by its arguments). The following SOUL snippet depicts a logic goal that consists of the binary predicate `contains:/2` and two terms. The first term is the logic variable *?collection* and the second is a so-called *Smalltalk term*:

```
1 ?collection contains: [1+3]
```

Such *Smalltalk terms* are delimited by square brackets and can contain logic variables wherever Smalltalk variables are allowed. Examples include `[3.4 asInteger]`, `[Object]` and `[?x > ?y]`. Logic lists are demarcated by angle brackets. Examples include the empty list `<>`, the list with three elements `<1,2,3>` and every list with head *?h* and tail *?t*: `<?h|?t>`. The expressions within *Smalltalk terms* are evaluated as standard Smalltalk, after logic variables have been substituted by the values they are bound to. As alluded to in the introduction, this mechanism is rather unsophisticated. In case *?x* has no binding, the Smalltalk VM will send the message `doesNotUnderstand:` to the Smalltalk object that implements the variable (i.e. a `Soul.Variable` instance). A Smalltalk term (e.g. the argument of a condition) unifies with another term (e.g. the parameter of the head of a rule) if its expression evaluates to a value that unifies with the term. A Smalltalk term can also be used as a condition on its own, in which case its expression has to evaluate to the singleton `True` in order for resolution to succeed. In practice, it is not necessary to know when the expression in a Smalltalk term is evaluated unless it has side-effects.[2]

The hybrid language characteristic of SOUL influences the design of its logic libraries. Predicate `contains:/2`, for instance, is equivalent to the `member/2` predicate of Prolog. In addition to the membership relation between logic lists and their elements, `contains:/2` captures the membership relation between Smalltalk collections and their elements. Variable bindings *?x* → 1, *?x* → 3 and *?x* → 5 are the solutions to the following query:

?y to 3, for instance, the first condition must have already bound *?y* to 1. Section 5.1.1 introduces the syntax of SOUL.

[2] Although carelessly crafted SOUL programs also incur a performance overhead from the repeated unification of side-effect free Smalltalk terms.

```
1 +?collection contains: ?element if
2   ?iterator equals: [?collection iterator],
3   [?iterator hasNext],
4   ?iterator iteratorPointsTo: ?element

5 +?iterator iteratorPointsTo: ?element if
6   ?element equals: [?iterator next].
7 +?iterator iteratorPointsTo: ?element if
8   [?iterator hasNext],
9   ?iterator iteratorPointsTo: ?element
```

Figure 5.1: Linguistic symbiosis with Java in the implementation of `contains:/2`.

```
1 if [1 to: 5 by: 2] contains: ?x
```

Linguistic Symbiosis with Java

The JAVACONNECT [Jav] library allows a Smalltalk application to invoke methods on any Java object in a running JVM instance. Java objects can be referenced through Smalltalk proxies. We have combined the existing symbiosis between SOUL and Smalltalk with the symbiosis between Smalltalk and Java provided by JAVACONNECT. This transitively establishes a symbiosis between SOUL and Java.

Figure 5.1 illustrates the resulting symbiosis. It depicts an additional clause for the `contains:/2` predicate of the SOUL standard library. This clause quantifies over all elements in a Java collection. The pre-existing clauses already quantified transparently over logic lists and Smalltalk collections.

On line 2, *?iterator* is bound to a Smalltalk proxy that wraps a Java iterator and forwards messages to it. Examples of such messages include `iterator` and `hasNext`. Note that line 2 introduces no choice-points and commits to a single iterator —recalling that care must be taken when using Smalltalk terms with side-effects. The `equals:/2` predicate is implemented by the fact *?x* `equals:` *?x*. It is equivalent to the `=/2` operator in Prolog.

Mode Annotations

SOUL supports mode annotations on variables in the head of a rule: the annotations on *+?var* and *-?var* indicate that the rule is only applicable when *?var* is bound and unbound respectively. Such annotations allow specializing the proof procedure for a predicate to its context of use. Predicate `isMethodDeclaration/1` from the CAVA library, for instance, can be used to quantify over all method declarations. It can also be used to check whether its argument is a method declaration. Checking whether the argument unifies with one of the method declarations would be inefficient for the second use. The implementation therefore discerns these two uses through mode annotations:

```
1 -?m isMethodDeclaration if
2   [Soul.MLI forJava allMethodDeclarations] contains: ?m
3 +?m isMethodDeclaration if
4   [?m isKindOf: JavaWorld.org.eclipse.jdt.core.dom.MethodDeclaration]
```

The Smalltalk expression `Soul.MLI forJava` evaluates to the meta level interface for Java, which is the central access point to the reified program representation. Note that the reified version of a method declaration is the method declaration itself (i.e. an instance of `org.eclipse.jdt.core.dom.MethodDeclaration` from the DOM of the Eclipse JDT Core Component [Ecl08a]). Linguistic symbiosis with Java allows the CAVA library (cf. Section 5.2) to forgo a transcription to com-

pound terms of `org.eclipse.jdt.core.dom.ASTNode` instances in its identity-based reification (cf. Section 4.4.2).

Variable Argument Terms

Other deviations from Prolog include support for variable argument compound terms (e.g. term `or@(?args)` as indicated by the `@` that precedes its argument list). Unifying the compound term `max(1,4,?x)` with the variable argument compound term `?functor@(?args)`, dissects the compound term in its functor `?functor`→`max` and arguments `?args`→`<1,4,?x>`[3].

Functor Variables

As illustrated above, logic variables can be used as functor and predicate symbols in SOUL. This higher-order syntax facilitates a limited form of higher-order programming. The higher-order `map/2` predicate, for instance, can be implemented as follows:[4]

```
1  map(<>,<>,?).
2  map(<?e1|?rest1>,<?e2|?rest2>,?predicate) if
3          ?predicate(?e1,?e2),
4          map(?rest1,?rest2,?predicate)

5  if map(<?x,2>,<1,?y>,#equals:)
6  if map(<1,2>,?list,[[:each | each + 3 ]])
```

The binding for the *?predicate* variable determines the actual goal that is resolved on line 3. Evaluating the query on line 5 results in bindings `?x`→`1` and `?y`→`2`. Keyword symbols can be used as functor symbols if the arity corresponds. Through such constructions, the translational semantics of example-based interpretations have been implemented in a generic manner (cf. Section 7.5). As demonstrated by the query on line 6 above, SOUL transparently supports instances of `BlockClosure` in the functor position on line 3 as well. In keeping with its hybrid characteristic, it will bind *?list* to the list `<4,5>`.

The next section presents a more formal account of the semantics through a meta-interpreter for SOUL.

5.1.2 Vanilla Meta-Interpreter for SOUL

Apart from minor deviations related to linguistic symbiosis, the proof procedure employed by SOUL is the same as the one employed by Prolog: SLDNF-resolution [Rob65, EK76]. Figure 5.2 depicts a vanilla meta-interpreter for SOUL. It differs only slightly from the traditional one for Prolog. We will revisit the meta-interpreter to illustrate how the fuzzy logic and example-based specification cornerstones extend SOUL. Figure 6.1, for instance, depicts the meta-interpreter corresponding to the fuzzy variant of SOUL.

On lines 1–5, the meta-interpreter demonstrates how SOUL resolves Smalltalk terms. On line 2, variable *&goal* is bound to a `Soul.SmalltalkTerm` instance. If

[3] This is equivalent to max(1,4,X)=..[Functor|Arguments] in Prolog.

[4] Using predicate `call/n` on line 3 would be more appropriate. This would support a goal `map([3,1],[X,Y],nth1([a,b,c,d]))` that binds X →c and Y →a. Predicate nth1/3 expects three arguments.

```
1  &goal isProven if
2    [&goal isKindOf: Soul.SmalltalkTerm],!,
3    getEnv(?env,?), envLookup(&goal,?gpointer),
4    ?value equals: [?gpointer term evaluateIn: ?env startAt: ?gpointer envIndex],
5    [?value = true].

6  ?goal isProven if
7    ?goal equals: and@(?goals),!,?goals isProvenListOfGoals.
8  ?goal isProven if
9    ?goal equals: or@(<&h|?t>),not(?t equals: <>),!,
10   or(&h isProven,or@(?t) isProven)
11 ?goal isProven if
12   ?goal equals: or(&h),!,&h isProven
13 ?goal isProven if
14   ?goal equals: not@(?goals),!,not(and@(?goals) isProven).

15 ?goal isProven if
16   ?goal isHeadOfRule: ? withBody: ?body,
17   ?body isProvenListOfGoals.

18 <> isProvenListOfGoals.
19 <&g|?r> isProvenListOfGoals if
20   &g isProven,
21   ?r isProvenListOfGoals
```

Figure 5.2: The vanilla meta-interpreter for SOUL.

a Smalltalk term is unified with a variable of the form *&var* rather than the regular *?var*, the variable is bound to the Smalltalk implementation of the term rather than the value of the expression within the term.

Lines 3–4 perform the actual evaluation of the expression within the Smalltalk term. Its value is bound to variable *?value*. It is computed by method `evaluateIn:startAt:` which evaluates an expression in which a single Smalltalk variable substitutes for all occurrences of a logic variable in the term. Each Smalltalk variable is assigned the binding of their corresponding logic variable in the current environment *?env*. A starting index in this environment is needed because SOUL optimizes variable lookup through lexical addressing. The reflective predicate `envLookup/2` binds its second argument to a wrapper (*?gpointer*) for the Smalltalk object that implements the binding for its first argument (*?gpointer* `term`) and its lexical index in the current environment (*?gpointer* `envIndex`).

Line 5 demonstrates that in order for the resolution of a Smalltalk term to succeed, its expression must evaluate to true.[5] The meta-interpreter for the fuzzy variant of SOUL will differ on this line.

Lines 6–14 clarify the semantics of SOUL's variable-argument connectives `and/n`, `or/n` and `not/n`.

Lines 15–21 illustrate resolution. To resolve a goal *?goal*, a list of goals *?goals* is resolved that corresponds to the body of a rule with a conclusion that unifies with *?goal*. Some essential ingredients of the proof procedure are not made explicit by the meta-interpreter. Candidate rule selection, backtracking and cuts are handled as in Prolog.

[5]Note that the symbol `true` is not assigned a special meaning in Soul. The query "*if* `true isProven`" will fail because there is no corresponding logic fact `true/0`. In contrast, the query "*if* `[true] isProven`" will succeed because of line 5 of the meta-interpreter.

5.2 CAVA: Predicates for Reasoning about Java Programs

We complemented the SOUL evaluator with a library of predicates for reasoning about Java programs: the CAVA library. Two types of predicates can be discerned: *reification predicates* and *basic reasoning* predicates.

In the LMP approach to pattern detection, the program representation is reified as values in the logic language such that variables can range over its information. The *reification predicates* implement relations that quantify over the reified program representation described in Section 4.2.3. This representation contains syntactic, structural, control flow and data flow information about the Java program. The *basic reasoning predicates* use the reification predicates to implement relations that are not explicit in the program representation. They can be used in specifications to implement more advanced relations that quantify over all instances of a pattern.

The next section details the reification predicates. The basic reasoning predicates are detailed in Section 5.2.2.

5.2.1 Reification Predicates

Our LMP instance uses an identity-based reification (cf. Section 4.4.2). This means that unifying logic terms are reified versions of the same AST node.[6] Linguistic symbiosis moreover enables forgoing the prevalent reification to compound terms: the reified version of an AST node (i.e. an `org.eclipse.jdt.core.dom.ASTNode` instance) is the AST node itself.

At any point in the proof procedure, this identity-based reification to objects renders reconstructing the actual AST node from its reified counterpart trivial. The AST node *is* the term (i.e. the object) at hand. The node's context within the program can be obtained through message sends. This facilitates querying whole-program analyses for the results of individual AST nodes —for instance, in the domain-specific unification procedure (cf. Section 6.7). Integration with other tools in the Smalltalk environment is facilitated as well. Query results consist of objects that can be used directly. The inspector windows on the right side of Figure 4.12, for instance, are standard tools in the Smalltalk environment.

The following query illustrates this reification:

```
1  if ?m isMethodDeclaration,
2     [?m modifiers isEmpty],
3     ?m methodDeclarationHasModifiers: ?list
```

Its solutions consist of methods that have been declared without modifiers. Upon backtracking over the first condition, *?m* gets bound successively to each `org.eclipse.jdt.core.dom.MethodDeclaration` instance in the program representation. Predicate `isMethodDeclaration/1` is one of the reification predicates defined in the CAVA library. The Smalltalk term on the second line filters out all method declarations that have modifiers. Message `modifiers` returns an instance of the Java list subclass `ASTNode$NodeList` (an innerclass of `org.eclipse.jdt.core.dom.ASTNode`). It answers message `isEmpty` with a Java boolean. Through an automatic conversion to the equivalent Smalltalk boolean, this answer determines whether the second condition succeeds.

[6] Provided the terms unify according to the general-purpose unification procedure.

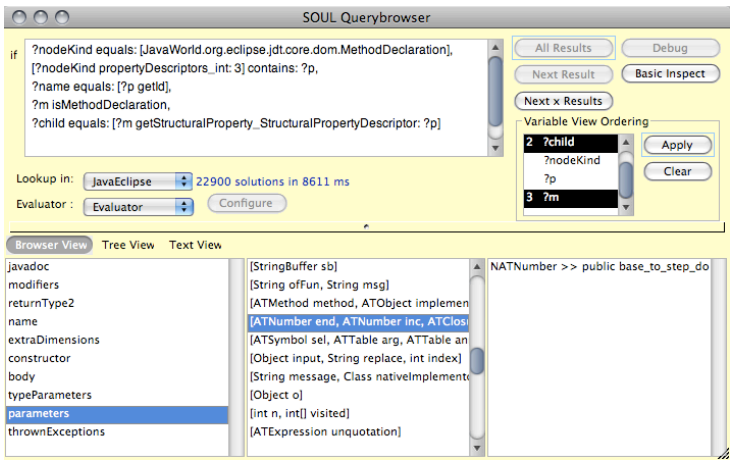


Figure 5.3: AST node meta-information enables generating reification predicates.

Predicate `methodDeclarationHasModifiers:/2`, used on the third line above, is another reification predicate. It reifies the relation between a reified method declaration and its reified modifier list. Note that the modifiers list *?list* of the method declaration is not converted to a logic list, but kept as an instance of `ASTNode$NodeList`. In other words, adding an additional condition `[?m modifiers = ?list]` would not change the results for the query.

The CAVA library defines reification predicates for the relations among the syntactic, structural and control flow information in the program representation.[7] The following sections detail these predicates.

Reification Predicates for Syntactic Information

Abstract syntax trees in the program representation stem from the DOM of the Eclipse JDT Core Component [Ecl08a]. Each node in these trees is an instance of an `org.eclipse.jdt.core.dom.ASTNode` subclass. In Figure 2.2, all AST nodes are depicted with their class and the concrete syntax elements they represent.

For each subclass of `ASTNode`, the CAVA library provides a unary predicate (e.g. `isMethodDeclaration/1`) that reifies all nodes of this kind in an Eclipse workspace. Binary predicates (e.g. `methodDeclarationHasModifiers/2`) reify the relations between each node and its children.

The reification predicates for syntactic information are generated automatically. This is possible because an API for structural reflection is implemented on the entire `ASTNode` hierarchy. Figure 5.3 illustrates the methods that provide information about the structure of the AST. The second condition retrieves a collection of property descriptors that describe the children of a `MethodDeclaration` node. The third condition retrieves the name of such a property descriptor *?p*. The fourth condition binds *?m* to a method declaration node (i.e. an instance of the class *?nodeKind* bound by the first condition). The last condition binds *?child* to the node's child that corresponds to descriptor *?p*. The query quantifies in a

[7] The results of the data flow analyses are not reified. This precludes them from popping up in solutions to a logic query (criterion **CDM1**). They will be used by the domain-specific unification procedure instead.

generic manner over all immediate children of all method declaration nodes. The first column of the results depicts all property descriptors for method declaration nodes. The second column depicts the values for the 'parameters' property (i.e. the child corresponding to the parameter list) of all method declarations in the Eclipse workspace.

The actual implementation of the reification predicates relies on the domain-specific unification of a reified program element with a structurally equivalent compound term (cf. Section 4.4):

```
1  ?m methodDeclarationHasModifiers: ?modifiers if
2    ?m isMethodDeclaration,
3    ?m equals: methodDeclaration(?,?modifiers,?,?,?,?,?,?,?)
```

The domain-specific unification procedure invokes the reflective API of the `ASTNode` hierarchy to map AST nodes to structurally equivalent compound terms. This leaves its implementation and the implementation of the reification predicates less brittle to changes in the parser and the language specification. The reification predicates evolve with the syntactic information they reify.

Reification Predicates for Structural Information

Our program representation includes structural information provided by the Java Model of the Eclipse JDT Core Component [Ecl08a]. [8] Figure 2.3 depicts the structural information available for a project `DissertationExample` in the Eclipse workspace. It offers information about the configuration of the project in the Eclipse workspace: its compilation units (i.e. source files), its compiled classes, the libraries it references, etc. The compiler needs this information to build the project. We use the same information to launch the data flow analyses of the program representation.

Structural information is also available for the types and methods declared within the project. Type `Example` is one of the types declared in compilation unit `Example.java`. Whether or not a type or method has a binary or source declaration is abstracted from. Note that the structural information does not include abstract syntax trees and is too coarse-grained to reconstruct the complete AST of the program.

The CAVA library provides predicates that reify the elements in the structural information and their relations. Predicate `isTypeWithFullyQualifiedName:/2`, for instance, reifies the relation between a type and its fully qualified name. Note that the exact binding for *?t* in "*?t* `isTypeWithFullyQualifiedName: ['java.lang.Thread']`" depends on the configuration of the project (i.e. the version of the standard library that is referenced by the project).

[8] Structural information can be derived from syntactic information. The logic rules that implement the `extends:/2` predicate in Figure 4.13, for instance, derive a type hierarchy from type declaration AST nodes. However, the AST for the declaration of a type is not always available (e.g. types imported from binary packages). Predicate `typeDeclarationForType:/2` (on lines 3 and 6 of the `extends:/2` implementation) fails on such types. It is therefore better to quantify over the structural information in the program representation.

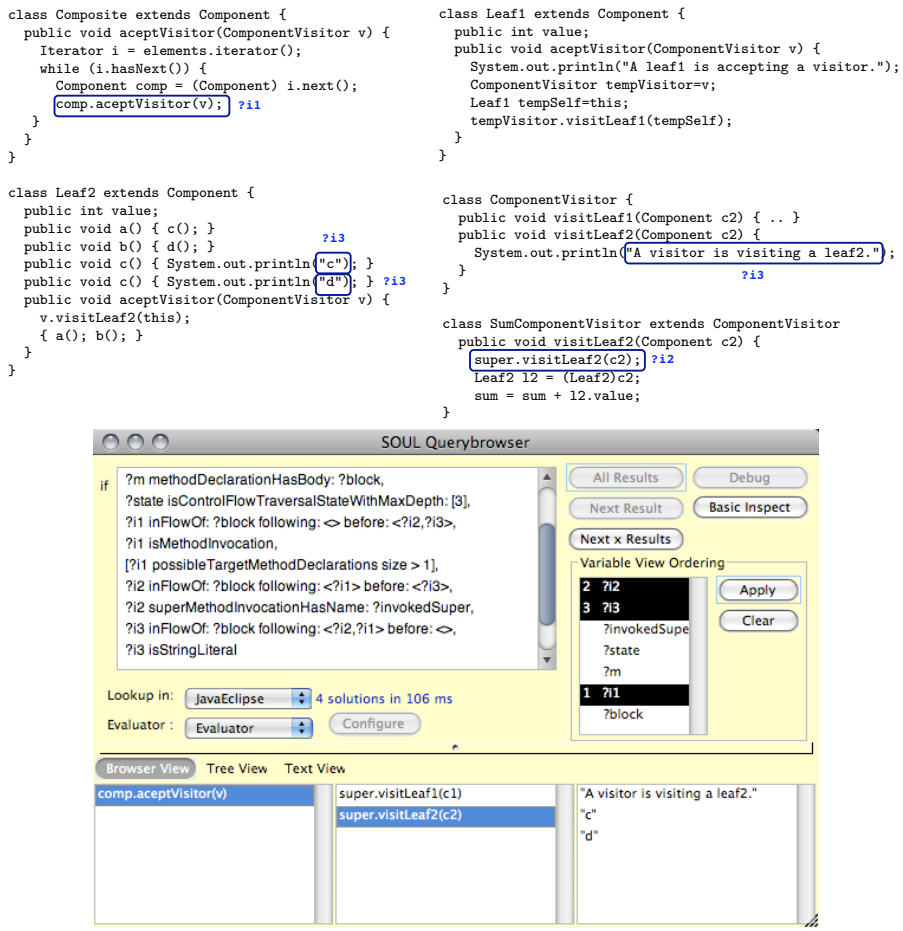


Figure 5.4: Illustrating reification predicates for control flow information.

Reification Predicates for Control Flow Information

The program representation includes an inter-procedural control flow graph that is computed on-the-fly (cf. Section 2.5.3). We implemented a method `nextNodeToBeMatched` on the `ASTNode` hierarchy. This method returns the node that, at run-time, would be executed after the receiver. A collection of nodes is returned in case the control flow splits after the receiver of `nextNodeToBeMatched`. Among others, this is the case for polymorphic method invocations. Computing the transitive closure of `nextNodeToBeMatched` enumerates all sequences of consecutively executed instructions (i.e. possible paths through the control flow graph).

The CAVA predicate `inFlowOf:following:before:/4` reifies information about the execution order of AST nodes by traversing the control flow graph. All of its arguments, except the first, are input arguments. Figure 5.4 illustrates its use. Solutions to the depicted query consist of three instructions *?i1*, *?i2* and *?i3* that may be executed consecutively at run-time. These instructions lie on the same path through the control flow graph. Unspecified instructions are allowed on this path (before, after and in between the specified instructions).

The path is computed by a depth-first traversal of the control flow graph. The third condition binds *?i1* to the first instruction in the control flow graph of *?block*. Upon backtracking over this condition (e.g. because the binding for *?i1* is not a method invocation as required by the fourth condition), *?i1* is bound to the next instruction in the graph. In case there are multiple successors to a node (i.e. a branch point in the graph), each path is followed completely until it is exhausted (i.e. depth-first traversal). Backtracking over an exhausted path will return to the latest branch point and start the traversal over a new path.

The fifth condition in the query explicitly requires *?i* to be a branch point in the graph: a method invocation that has multiple possible target method declarations (i.e. due to late binding and polymorphism). Using the results from the points-to analysis, these can be determined based on an approximation of the dynamic type of the receiver (i.e. the dynamic type of all heap object approximations in its points-to set) rather than its static type (e.g. class hierarchy analysis [DGC95]). In fact, the precision of points-to analyses is often compared using the amount of virtual method invocations they can resolve [LH06].

The first column of Figure 5.4 depicts the method invocation in the flow of method `Composite.aceptVisitor(ComponentVisitor)`[9] that could not be resolved completely. After this instruction, the control flow splits. One path corresponds to an invocation of method `Leaf1.aceptVsitor(SumComponentVisitor)` while the other corresponds to an invocation of method `Leaf2.aceptVisitor(SumComponentVisitor)`. The second column depicts a different binding for *?i2* on each path. Each binding corresponds to a different super invocation in `SumComponentVisitor`. The third column depicts string literals that are evaluated on the path after the invocation of `super.visitLeaf2(c2)` (i.e. bindings for *?i3*).

Each method invocation is followed once —even if the intra-procedural control flow graph of the target method declaration has already been traversed. A method invocation is not returned from until all paths through the target declaration have been exhausted. The resulting analysis is therefore inter-procedural and context-sensitive. Cycles in the graph are followed once. The maximum depth of the simulated call stack can be customized. In the query, the second condition limits the stack to three invocations.

Compared to state of the art algorithms for evaluating regular path expressions over control flow graphs (cf. Section 3.4), the CAVA predicates only perform straightforward graph traversals. The third condition in the query, for instance, does not generate bindings for variables *?i2* and *?i3*. If these variables were bound, however, the predicate would backtrack to the latest branch point when they are encountered on a path. They represent nodes on the path to stay ahead, but the traversal never checks whether they actually follow the bindings for its first argument.

5.2.2 Basic Reasoning Predicates

CAVA provides basic reasoning predicates that use the reification predicates introduced above. They implement relations between program elements that are not explicit in the program representation.

Basic reasoning predicate `isChildOf:/2`, for instance, can be used to traverse

[9] The spelling error is deliberate.

ASTs. [10] It is implemented by the following rules:

```
1 ?term isChildOf: ?functor@(?args) if
2   ?args contains: ?child, ?term isChildOf: ?child
3 ?term isChildOf: ?term if
4   not([?term isKindOf: JavaWorld.org.eclipse.jdt.core.dom.ASTNode_NodeList])
```

The rules rely on the domain-specific unification between an AST node and a structurally equivalent compound term (cf. Section 4.4). Unifying an AST node with the variable argument compound term `?functor@(?args)` (cf. Section 5.1.1) binds *?args* to the children of the node. The first rule recurses over these children. The second rule is the stop condition.[11]

Other basic reasoning predicates implement the relations between the syntactic information and the structural information in the program representation. Consider predicates `declaresType:/2` and `extendsType:/2`. The former implements the relation between a type declaration AST node and the type it declares. The latter implements the relation between a type declaration AST node and the type it extends. Such types stem from the structural rather than the syntactic information in the program representation. This is because a type that is referred to in the source code, may be declared in any of the byte code libraries included in the base program. No syntactic information is available for such types (i.e. an AST node for their declaration). In general, this is the case for the types declared in the java standard library (e.g. `java.lang.Thread`).

Figure 5.5 depicts the results for a query that uses predicate `extendsType:/2` to quantify over all class declaration AST nodes (first condition) that extend an immediate super type (second condition) for which no source code is available (third condition). The third column depicts the class declaration itself. The first column depicts all fully qualified names of these types (i.e. bindings for *?name*). The second column depicts the bindings for *?superNode*. This is the AST node that corresponds to the concrete syntax elements after the `extends` keyword in the class declaration. Because of the identity-based reification of AST nodes, there are multiple entries in the second column for each entry in the first column.

The rules that implement the `extendsType:/2` predicate rely on the results of the semantic analysis to map the AST node *?superNode* (corresponding to the concrete syntax elements after the `extends` keyword) to a binary or source type *?type*. These results are necessary because the actual type referred to by the AST node depends on its context of use in the program (e.g. the import declarations of the compilation unit it resides in). The domain-specific unification procedure will unify type declaration nodes with type nodes based on the same semantic analysis. This way, users can benefit from its results without being exposed to its details.

[10] Note that `isChildOf:/2` does not manage a traversal context. The context within the AST of each returned node can be queried by invoking methods on the node (e.g. method `parentOfKind:avoiding:`). Of course, there is a computational overhead associated with reverse tree walks that follow each descend into a node. The alternative is using a higher-order traversal predicate that passes its traversal context to user-provided predicate arguments (e.g. predicate `traverseMethodParseTree(?method,?result,?found,?process)` in Figure 4.5).

[11] The elements in an `ASTNode$NodeList` are returned by the first rule. A `NodeList` instance unifies with compounds `nodeList(?elements)`.

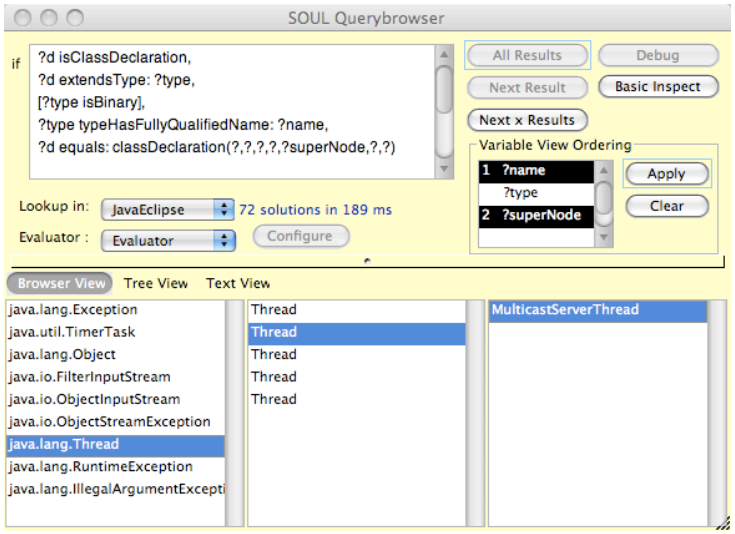


Figure 5.5: Classes with an immediate super-type for which no AST is available.

5.3 LMP Support for Pattern Characteristics

Having introduced SOUL and the CAVA library, we will now demonstrate how this instantiation of the LMP cornerstone supports each of the pattern characteristics identified in Section 2.2:

Section 5.3.1 specifies and detects the *syntactic characteristics* of regular `for`-statements that can be transformed to the enhanced `for`-statements introduced in Java 1.5.

Section 5.3.2 specifies and detects the *structural characteristics* of application-specific coding conventions.

Section 5.3.3 specifies and detects the *control flow characteristics* of the protocol an API expects to be adhered to.

Section 5.3.4 specifies and detects the *data flow characteristics* of the aforementioned protocol and enhanceable `for`-statements.

We will point out the unification-related (cf. Section 4.4.2) and quantification-related (cf. Section 4.2.2) shortcomings of regular LMP as manifested in these examples. The examples will therefore be revisited in future chapters to show how these shortcomings are remedied by the other cornerstones of our approach.

We will furthermore specify the patterns with non-syntactic characteristics twice. Once through the predicates that reify the information that supports them (e.g. control flow characteristics through reification predicates for control flow information) and once through the reification predicates for syntactic information. We will show that the latter specifications are far from descriptive, have recurring parts and possibly lead to a lower recall and false positives —motivating the importance of explicit support for each characteristic.

5.3.1 Expressing Syntactic Characteristics

In this section, we specify and detect the syntactic characteristics of regular `for`-statements that could be transformed to the enhanced `for`-statement introduced in Java 1.5. To enumerate all elements in a collection, a regular `for`-statement that iterates through the collection can be used:

```
1 for(Iterator i = list.iterator(); i.hasNext();) {
2    Object element = i.next();
3 }
```

Alternatively, such enumerations can be implemented using an enhanced `for`-statement of the form:

```
1 for (Object element : list) {
2 }
```

The logic query at the top of Figure 5.6 specifies the syntactic characteristics of potentially enhanceable `for`-statements. The query uses the reification predicates for syntactic information (cf. Section 5.2.1) and the basic reasoning predicate `isChildOf:/2`. The latter predicate implements an AST traversal of its second argument (cf. Section 5.2.2).

The first four lines of the query quantify over all `for`-statement AST nodes and their expression, updaters and body children. They establish the binding *?condition* →`i.hasNext()` for the enhanceable `for`-statement above. Lines 5–10 traverse the *?condition* expression of the `for`-statement to retrieve an invocation of method `hasNext()`. Variable *?hasNextReceiver* is bound to the receiver of this invocation. Requiring the invocation to be a child of the expression rather than the expression itself, ensures that more potentially enhanceable statements are recognized at the cost of necessitating a manual assessment of the solutions to the query. Lines 11–16 require an invocation of method `next()` to reside in the body of the `for`-statement or in one of its updaters.

The bottom-left corner of Figure 5.6 depicts the outcome of this query on the program depicted in the bottom-right corner. The binding for *?method* is established by an additional condition `[`*?for* `parentMethodDeclaration] equals:` *?method*. In addition to the conditions on lines 10 and 16, this condition illustrates our identity-based reification to objects: the binding of *?for* is queried for its method declaration parent through a method invocation.

The query identifies all potentially enhanceable `for`-statements. It also reports the false positive in method `not_enhanceable_1`. This is because the query does not specify a data flow characteristic stating that the receivers of the invocations of `hasNext()` and `next()` should be the same iterator. Section 5.3.4 discusses how this characteristic can be expressed. Note that method `enhanceable_3` features three times in the solutions. The second solution is a false positive. Its binding *?nextReceiver* →`j` does not correspond to its binding *?for* →`for(Iterator i = l.iterator(); i.hasNext();)`. The binding for *?nextReceiver* in this solution originates from the inner loop rather than the outer loop.

Evaluation The query uses an AST traversal on line 11 to express that invocation node *?nextInv* can reside at an arbitrary depth within the *?body* node of the `for`-statement. The aforementioned false positive can only be eliminated by further

```
if ?for isStatement,
   ?for forStatementHasExpression: ?condition,
   ?for forStatementHasUpdaters: ?updaters,
   ?for forStatementHasBody: ?body,

   ?hasNextInv isChildOf: ?condition,
   ?hasNextInv methodInvocationHasName: ?hasNextName,
   ?hasNextName simpleNameHasIdentifier: ['hasNext'],
   ?hasNextInv methodInvocationHasExpression: ?hasNextReceiver,
   ?hasNextInv methodInvocationHasArguments: ?hasNextArguments,
   [?hasNextArguments size = 0],

   or(?nextInv isChildOf: ?body,?nextInv isChildOf: ?updaters),
   ?nextInv methodInvocationHasName: ?nextInvName,
   ?nextInvName simpleNameHasIdentifier: ['next'],
   ?nextInv methodInvocationHasExpression: ?nextReceiver,
   ?nextInv methodInvocationHasArguments: ?nextArguments,
   [?nextArguments size = 0],
```

```
for -> for (; i.hasNext(); ) {
hasNextReceiver -> i
nextReceiver -> ((Iterator)temp)
method -> ■ ForStatements >> enhanceable_4()
for -> for (Iterator j=l.iterator(); j.hasNext(); ) j.next();
hasNextReceiver -> j
nextReceiver -> j
method -> ■ ForStatements >> enhanceable_3()
for -> for (; i.hasNext(); ) j.next();
hasNextReceiver -> i
nextReceiver -> j
method -> ■ ForStatements >> not_enhanceable_1()
for -> for (Iterator i=l.iterator(); i.hasNext(); ) {
hasNextReceiver -> i
nextReceiver -> j
method -> ■ ForStatements >> enhanceable_3()
for -> for (Iterator iterator=l.iterator(); iterator.hasNext(); )
hasNextReceiver -> iterator
nextReceiver -> iterator
method -> ■ ForStatements >> enhanceable_1()
for -> for (Iterator i=l.iterator(); i.hasNext(); ) {
hasNextReceiver -> i
nextReceiver -> i
method -> ■ ForStatements >> enhanceable_3()
for -> for (Iterator j=i; i.hasNext(); ) {
hasNextReceiver -> i
nextReceiver -> j
method -> ■ ForStatements >> enhanceable_2()
```

```
public class ForStatements {
  public List l = new ArrayList();
  void enhanceable_1() {
    for (Iterator iterator = l.iterator();
         iterator.hasNext();) {
        iterator.next();
    }
  }

  void enhanceable_2() {
    Iterator i = l.iterator();
    for (Iterator j = i; i.hasNext();) {
         j.next();
    }
  }

  void enhanceable_3() {
    for (Iterator i = l.iterator(); i.hasNext();) {
      Object o = i.next();
      for (Iterator j = l.iterator(); j.hasNext();)
           j.next();
    }
  }

  void enhanceable_4() {
    Iterator i = l.iterator();
    Object temp = i;
    for (; i.hasNext();) {
      ((Iterator) temp).next();
    }
  }

  void not_enhanceable_1() {
    Iterator i = l.iterator();
    Iterator j = l.iterator();
    for (; i.hasNext();)
        j.next();
    }
  }
}
```

Figure 5.6: LMP specification for syntactic char. of enhanceable `for`-statements.

restraining the nesting relation between both nodes. For instance, by not descending into inner `for`-statements within the body of an outer `for`. This solution evidences the *quantification-related shortcomings* of LMP (cf. Section 4.2.2) through the higher-order traversal predicate and managing of traversal contexts it requires. Our identity-based reification to objects allows an alternative solution. The following condition could be added to eliminate the false positive:

```
1 [?hasNextInv parentOfKind: org.eclipse.jdt.core.dom.ForStatement] equals: ?for
```

However, this solution is equally operational in nature. Moreover, it introduces a computational overhead because a reverse tree walk is performed for each *?hasNextInv*.

The query relies on predicates that reify AST nodes and predicates that reify the relation of each AST node with its child nodes (e.g. `isForStatement/1` and `forStatementHasBody:/2`). The convoluted sequences of the latter (e.g. the "... has ..." conditions on lines 2–4, 6–10, 12–16) are necessary because the general-purpose unification procedure does not unify reified AST nodes with structurally equivalent compound terms. These sequences illustrate the *unification-related shortcomings* of an identity-based reification combined with the general-purpose unification procedure (cf. Section 4.4.2).

5.3.2 Expressing Structural Characteristics

The specifications presented so far quantified directly over the relations among program elements that are made explicit by the reification predicates. Logic rules allow users to derive additional, application-specific relations among program elements from the ones that are reified. We will demonstrate this using rules that are specific to the 2008/02/01 implementation of the interpreter for the AMBIENTTALK [Amb] programming language. Appendix B.1 details some statistics about this program. In particular, we will specify and detect the structural characteristics of violations against two AMBIENTTALK-specific coding conventions. The first specification expresses the structural characteristics in terms of syntactic characteristics, while the second specification expresses them directly.

Expressing Structural Characteristics in Terms of Syntactic Characteristics

The implementation of the AMBIENTTALK interpreter has a hierarchy of interfaces that extend a root interface `ATObject`. Within this hierarchy, all methods of which the name starts with prefix `base_` or `meta_` are called "native methods". These methods have to adhere to an AMBIENTTALK-specific coding convention: their return type and the types of their parameters should be declared in the `ATObject` interface hierarchy. This avoids that method signatures refer to concrete classes that implement an abstract interface.

The logic rules at the top of Figure 5.7 define application-specific predicates `isATObjectInterface/1` and `isNativeATMethodDefinedIn:/2`. They quantify over the interfaces in the `ATObject` hierarchy and their native methods respectively. The queries at the bottom of Figure 5.7 refer to the predicates defined by these rules. As a result, they did not have to duplicate the conditions in the bodies of the rules.

The bottom-left query can be used to check whether all "native methods" adhere to the required convention. For instance, by manually com-

application-specific structural characteristics

```
1  ?interface isATObjectRootInterface if
2    ?interface interfaceDeclarationHasName: ?name,
3    ?name simpleNameHasIdentifier: ['ATObject']

4  ?interface isATObjectInterface if
5    ?root isATObjectRootInterface,
6    or(?interface equals: ?root,?interface interfaceExtends: ?root)

7  ?m isNativeATMethodDefinedIn: ?t if
8    ?t isATObjectInterface,
9    ?t definesMethod: ?m,
10   ?m methodDeclarationHasName: ?name,
11   ?name simpleNameHasIdentifier: ?id,
12   or(['base_*' match: ?id],['meta_*' match: ?id])
```

query that checks the prescribed coding convention

```
if ?m isNativeATMethodDefinedIn: ?i,
   ?m methodDeclarationHasReturnType: ?returnType,
   ?decReturnType typeDeclarationForType: ?returnType,
   ?m methodDeclarationHasParameters: ?pars,
   ?decReturnType isATObjectInterface,
   forall(?pars contains: ?p,
          and(?p singleVariableDeclarationHasType: ?parType,
              ?decParType typeDeclarationForType: ?parType,
              ?decParType isATObjectInterface))
```

query that detects violations against the coding convention

```
if ?m isNativeATMethodDefinedIn: ?i,
   ?m methodDeclarationHasReturnType: ?returnType,
   ?decReturnType typeDeclarationForType: ?returnType,
   ?m methodDeclarationHasParameters: ?pars
   ?pars contains: ?p,
   ?p singleVariableDeclarationHasType: ?parType,
   ?decParType typeDeclarationForType: ?parType,
   or(not(?decParType isATObjectInterface),
      not(?decReturnType isATObjectInterface))
```

Figure 5.7: LMP specification for structural characteristics of a coding convention.

paring its solutions with the solutions for the first condition only. Violations of the coding convention can also be detected by negating all but the first condition in the query: `if ?m isNativeATMethodDefinedIn:?i, not(and(?m methodDeclarationHasReturnType:?returnType,...))`. Alternatively, the bottom-right query can be used to detect such violations.

Evaluation The above specifications illustrate the facilities for abstraction and reuse (criterion **CSL4**) provided by the LMP cornerstone (e.g. reuse of user-defined predicate `isATObjectInterface/1`) and its facilities for expressing explicit points of variation (criterion **CLS3**) among pattern instances (e.g. logic connective `or/n` on line 12).

The specifications are descriptive, but convoluted due to the unification-related shortcomings of an identity-based reification combined with the general-purpose unification procedure (cf. Section 4.4.2). If it were possible to unify `SimpleName` instances with structurally equivalent compound terms, for instance, the conditions that extract the identifier from the name of an interface declaration (lines 2–3) could be replaced by a single and more descriptive `?interface hasName: simpleName(?identifier)` condition. The same goes for the conditions that extract the identifier from the name of a method declaration (lines 10–11).

Finally, the specifications express structural characteristics by quantifying over the reification predicates for syntactic information (cf. Section 5.2.1). This might lead to false positives. Predicate `isAtObjectRootInterface/1`, for instance, will confuse interfaces with unqualified name `ATObject` in a package that differs from `edu.vub.at.objects` with the root of the AMBIENTTALK interface hierarchy. Moreover, the predicate only succeeds if the AST for the root interface declaration is available. This need not be the case as the base program can include its bytecode instead. Using reification predicates for structural information would have avoided these problems.

```
1  if ?rootType isTypeWithFullyQualifiedName:
2              ['edu.vub.at.objects.natives.NativeATObject'],
3     or(?class inClassHierarchyOfType: ?rootType,
4        ?class typeDeclarationForType: ?rootType),

5     ?class definesMethod: ?m,
6     ?m methodDeclarationHasName: ?name,
7     ?name simpleNameHasIdentifier: ?id,
8     [?id matchesRegex: '(meta|base)_.+'],

9     not(and(?class inClassHierarchyOfType: ?super,
10            ?super definesMethod: ?superm,
11            ?m overrides: ?superm)),

12    not(and(or(?class implementsType: ?interfacetype,
13               and(?class inClassHierarchyOfType: ?supertype,
14                   ?superclass declaresType: ?supertype,
15                   ?superclass implementsType: ?interfacetype)),
16            ?interface declaresType: ?interfacetype,
17            ?interface definesMethod: ?interfacem,
18            ?m overrides: ?interfacem))
```

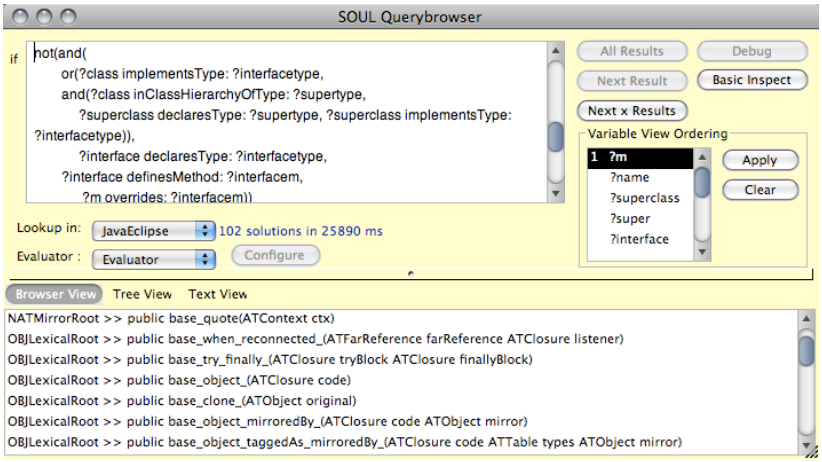


Figure 5.8: LMP specification for the structural char. of violations of a convention.

Expressing Structural Characteristics Directly

A closely related AMBIENTTALK-specific coding convention concerns the class hierarchy with root `NativeATObject` which parallels the interface hierarchy with root `ATObject`. Classes that define their own native methods (i.e. methods with the `base_` or `meta_` prefix that are not defined in a super class) should implement an interface in which those methods are defined. It can also be the case that the interface is implemented by a super class.

The specification at the top of Figure 5.8 can be used to detect violations of this coding convention. Lines 1–4 identify class declarations in the `NativeATObject` hierarchy. They are similar to the `isATObjectInterface/1` definition given above. Lines 5–8 identify the native methods defined by each class in the hierarchy. Lines 9–11 filter out the methods that override a method from a super class in

the hierarchy.[12] Lines 12–18 filter out methods that adhere to the coding convention. Lines 12–16 identify the interface that declares the type that is implemented either by the class or one of its super classes. Lines 17–18 identify methods from the class that override a method from this interface.

The violations depicted at the bottom of Figure 5.8 have been corrected in more recent implementations of the AMBIENTTALK interpreter.

Evaluation The above specification expresses structural characteristics through reification predicates for structural information (cf. Section 5.2.1). This avoids false positives with respect to these characteristics. On line 2, for instance, predicate `isTypeWithFullyQualifiedName:/2` will not confuse classes with unqualified name `NativeATObject` in a package that differs from `edu.vub.at.objects.natives` with the root of the AMBIENTTALK class hierarchy.

The specification also uses basic reasoning predicates that relate the syntactic and structural information in the program representation (cf. Section 5.2.2). The first and second argument to the `implementsType:/2` goal on line 12, for instance, are an AST node and a source or binary type from the structural program information respectively.

If we had expressed the structural characteristics of this coding convention using the reification predicates for syntactic information, we had to derive the `implementsType:/2` relation ourselves and consider the context of each syntactic element correctly. This is not always straightforward. The actual type implemented by a class declaration AST node, for instance, is determined by the import declarations of the compilation unit in which it resides. We would therefore have to find this compilation unit through a backwards AST traversal —leading to operational and possibly flawed pattern specifications.[13]

5.3.3 Expressing Control Flow Characteristics

This section evaluates the support for control flow characteristics offered by regular LMP. Concretely, we apply SOUL and the CAVA library to another application of user-specified pattern detection: checking conformance with and violations of the protocol of an API (cf. Section 2.3).

An invocation of method `a()` initiates the protocol. This invocation should be followed by an invocation of method `c(Object)`. This invocation should take the result returned by `a()` as its argument. Between the invocation of a and c, there should not be an invocation of method `b()`. The bottom-left corner of Figure 5.9 depicts the method declarations that correspond to these invocations.

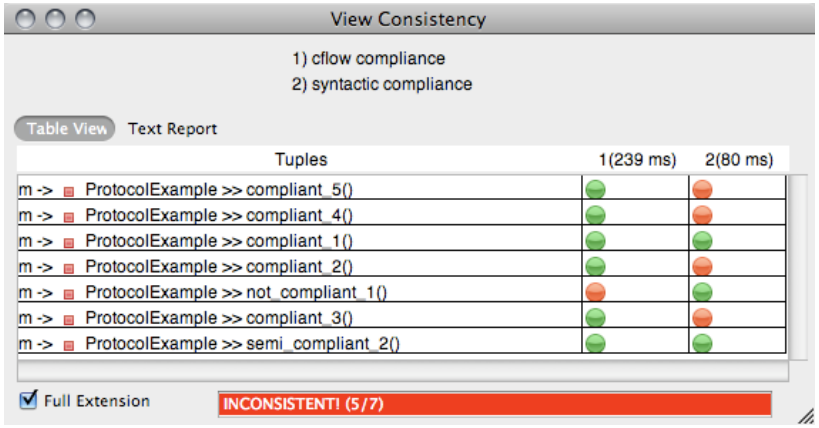
We will express the control flow characteristics of methods that comply with the protocol and of methods that violate the protocol. Figure 5.9 depicts examples of both. In Section 5.3.4, we will discuss how to express their data flow characteristics.

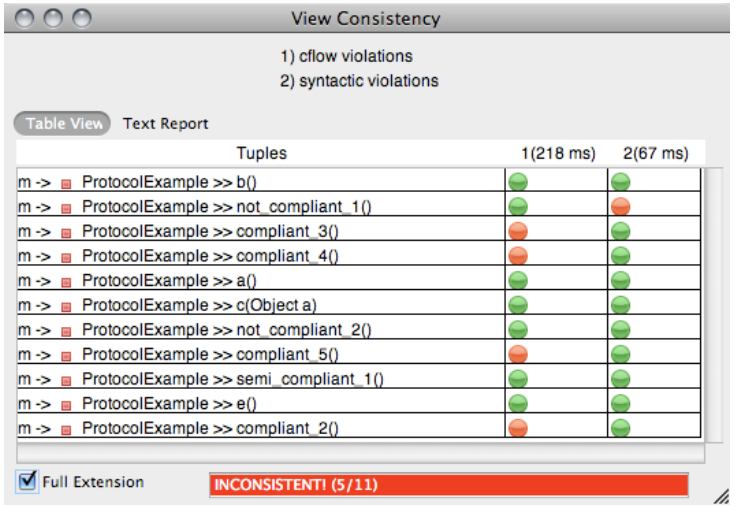
Expressing Control Flow Characteristics in Terms of Syntactic Characteristics

The query in the top-left corner of Figure 5.10 expresses the control flow characteristics of methods that comply with the protocol. The conditions on lines 1–5 select

[12] A single condition `not(?m overrides: ?)` would have sufficed.

[13] Alternatively, we could consult the semantic analysis results in the program representation. After all, this is the analysis that ensures the correctness of the basic reasoning predicates. However, criterion **CSL5** requires the specification language to hide the intricate details of this analysis.






```
class ProtocolExample {
        Date a() { return new Date(); }
        void b() { }
        Date c(Object a){ return (Date) a; }
        Object e() { return a(); }
        Date temp;

        void compliant_1() {
               Date d = a();
               c(d);
               b();
        }
        void compliant_2() {
               b();
               c(a());
        }
        void compliant_3() {
               c(e());
        }
```

```
void compliant_4() {
       temp = a();
       if(false) {
              b();
       } else {
              c(temp);
       }
}
class ProtocolCaller {
        void call() {
               c(a());
        }
}
void compliant_5() {
        new ProtocolCaller().call();
}
void semi_compliant_1() {
       temp = a();
       if(false) {
              b();
       }
       c(temp);
}
```

```
void semi_compliant_2() {
       temp = a();
       if(false) {
              c(temp);
       }
       b();
}
void not_compliant_1() {
        Date d;
        if(true) {
               d = a();
        } else {
               c(d);
        }
}
void not_compliant_2() {
        Date d = a();
        System.out.println(d);
        b();
        c(d);
}
```

Figure 5.9: Results for the queries that check protocol conformance depicted in Figure 5.10.

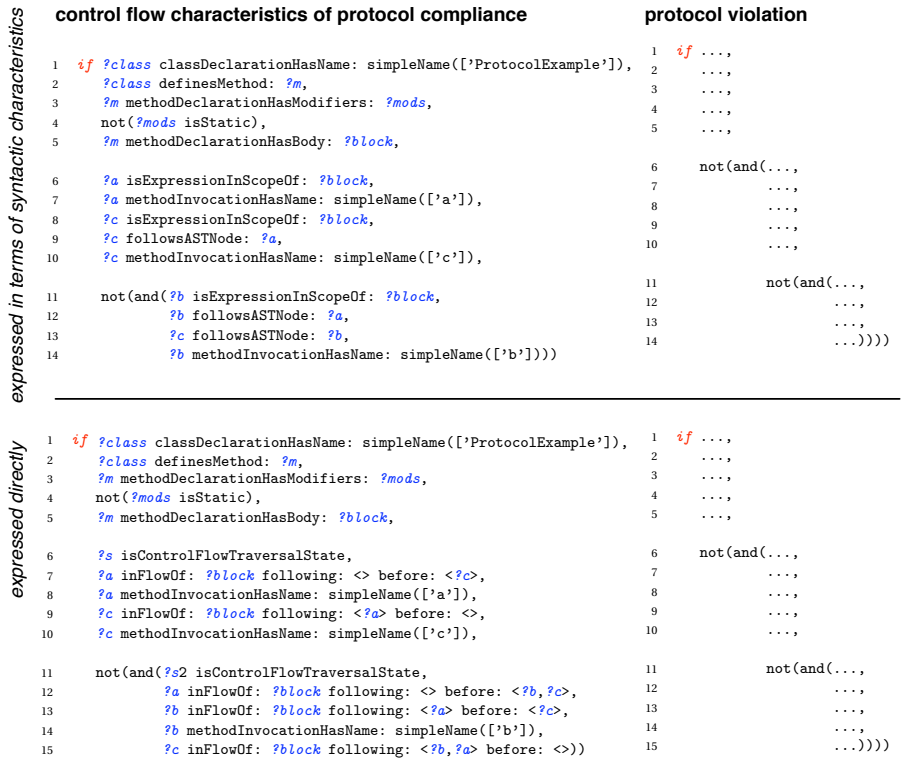


Figure 5.10: LMP specifications for protocol-related control flow characteristics.

the body *?block* of instance methods defined in the class `ProtocolExample`. Lines 6–10 state that an invocation of a method named `c` should follow the invocation of a method named `a`. Lines 11–14 express that there should not be an invocation of a method named `b` in between.

The query expresses the control flow characteristics in terms of syntactic characteristics. Predicate `isExpressionInScopeOf:/2` is equivalent to an AST traversal of the second argument in search of expressions.[14] Variables *?a* (line 6) and *?c* (line 8) are thus bound to invocations in the body of the method. The invocation of method `c` is required to follow the invocation of method `a`. This is expressed using predicate `followsASTNode:/2` on line 9. It is defined as follows:

```
1 +?astnode1 followsASTNode: +?astnode2 if
2   ?pos1 equals: [?astnode1 getStartPosition],
3   ?pos2 equals: [?astnode2 getStartPosition],
4   [(?pos1 > ?pos2) and: [?pos1 > (?pos2 + (?astnode2 getLength))]]
```

The predicate determines whether the execution of *?astnode1* follows the execution of *?astnode2* using the positions in the source code of the concrete syntax elements they represent (whitespace not included). This implementation leads to instances

[14] The condition on line 6 is equivalent to `?a isChildOf: ?block, ?a isExpression`. The implementation of the predicate differs out of performance considerations. Its results are cached. The predicate is used in queries generated under the lexical interpretation for template terms.

of the control flow characteristic being missed and false positives being reported. The positions in the source code can only be used as a rough indication of the actual run-time execution order (cf. Section 2.4.3). This is illustrated by the solutions to the query. They are depicted in the right column of the top-left window in Figure 5.9.[15] A green entry indicates that the method in the row label is included in the solutions (i.e. complies with the protocol according to the query). Red entries indicate that the method is not included in the solutions.

The solutions to the query consist of methods `compliant_1()`, `semi_compliant_2()` and `not_compliant_1()`. Only the first two methods comply with the control flow characteristics of the protocol in reality.[16] The other complying methods are not identified. Method `compliant_2()`, for instance, is not identified because *?a* lies within *?c*. Method `not_compliant_1()` is incorrectly identified by the query because *?c* follows *?a* in the source code —but not at run-time. The same goes for method `semi_compliant_2()`.

Violations of the protocol can be detected by negating the conditions on lines 6–14. This is illustrated by the query in the top-right corner of Figure 5.10. Its solutions correspond to the green entries in the right column of the top-right window depicted in Figure 5.9. The query does not recognize the aforementioned methods `compliant_1()`, `semi_compliant_2()` and `not_compliant_1()` as violations of the protocol. They are either not listed in the table (because the other query did not recognize them either) or they have a red entry in the column. All other methods are incorrectly recognized as violations.

Evaluation Detecting control flow characteristics using a tool that does not explicitly support them is hard. Users of such a tool have to derive information about the order in which instructions can be executed at run-time. The above specifications attempted to approximate this information in an ad-hoc manner which lead to false positives and a lower recall. Clearly, line numbers only give a rough indication of the actual execution order. The alternative, constructing a precise control flow graph, requires users to take the complete semantics of the programming language into account. This is far from trivial (cf. Section 2.5.3). Criterion **CSL1** therefore requires the specification language to explicitly support specifying control flow characteristics.

Expressing Control Flow Characteristics Directly

The query in the bottom-left corner of Figure 5.10 expresses the control flow characteristics of complying methods directly. It uses CAVA predicates that traverse a control flow graph (cf. Section 5.2.1).

Lines 1–5 are identical to the previous query. Line 6 initiates a new series of control flow traversals. Traversal state *?s* can be thought of as an implicit parameter to the occurrences of predicate `inFlowOf:following:before:/4` on lines 7 and 9. It keeps track of the nodes that have already been visited. Line 7 traverses the control flow graph in search for a binding for *?a*. Note that variable *?c* is unbound on line 7. Variable *?c* can therefore not function as a boundary in the control flow

[15] The left column depicts the solutions to a query that specifies the control flow characteristics directly (i.e. the query at the bottom of Figure 5.10). This way, the solutions to both queries can be easily compared.

[16] Method `semi_compliant_2()` complies with the protocol when the condition of its `if`-statement evaluates to `true`. Note that we do not consider data flow characteristics in this section.

graph for *?block*. Line 7 will not establish any bindings for this variable either.[17] Line 9 traverses the graph starting at *?a* in search for a binding for *?c*. It expresses that *?c* should follow *?a* in the control flow of *?block*.

Lines 11–15 state that there should not be an invocation of a method named `b()` in between the execution of *?a* and *?c*. Line 11 initiates a new series of traversals. This time, *?a* and *?c* are already bound. Line 12 will therefore dismiss bindings for *?a* that cross the boundary set by *?c*. Likewise, line 13 traverses *?block* between *?a* and *?c* in search for a binding for *?b*.

The solutions to the query are depicted in the left column of the top-left window in Figure 5.9. Again, green entries indicate that a method is included in the solutions. The query recognizes all methods that comply with the protocol. Its solutions do not include the false positive `not_compliant_1`. Note that the control flow graph traversal had to cross method boundaries to recognize methods `compliant_3` and `compliant_5` (i.e. it is inter-procedural). We will discuss methods `semi_compliant_1` and `semi_compliant_2` later.

Negating the conditions on lines 6–15 results in the query depicted in the bottom-right corner of Figure 5.10. It detects methods that violate the protocol. Its solutions correspond to the green entries in the left column of the top-right window depicted in Figure 5.9. The query recognizes all violating methods correctly. None of the complying methods are included in its results.

Method `semi_compliant_1` is recognized as a method that violates the protocol. There are two execution paths through the method. One in which the condition of the `if`-statement evaluates to `true` and one where it evaluates to `false`. The traversal predicate assumes that both are possible. The method complies with the protocol on the second path. However, the final conditions of the query specify that there should not be any traversal of the method that retrieves an invocation of `b()`. It is therefore reported as a violation of the protocol.

Method `semi_compliant_2` is, on the other hand, recognized as a method that complies with the protocol. There is an execution path through the method on which `c` is executed after `a`. However, there is also an execution path on which `c` is never executed. This is because lines 6–10 of the query are existentially qualified.

Evaluation Compared to the specification languages of tools that are tailored to control flow characteristics (cf. Section 3.4), CAVA's graph traversal predicates comprise a convoluted means to express such characteristics. The example-based specification cornerstone will remedy this quantification-related shortcoming of LMP (cf. Section 4.2.2). It enables exemplifying control flow characteristics through code excerpts.

In essence, successive control flow traversals express an existential path query. Negating a series of successive control flow traversals renders the path query universal. It is therefore not possible to express an existential path query in which all but a certain instruction is allowed. Moreover, successive control flow graph traversals cannot attain the performance levels of state of the art algorithms for evaluating path queries.

We will discuss these limitations of the control flow traversal predicates in Section 5.5.2. The example-based specification cornerstone provides a more descriptive means to express control flow characteristics, but its instantiation in our proto-

[17] Predicate `inFlowOf:following:before:/4` is a straightforward graph traversal predicate. It does not perform model checking of the graph (cf. Section 5.2.1).

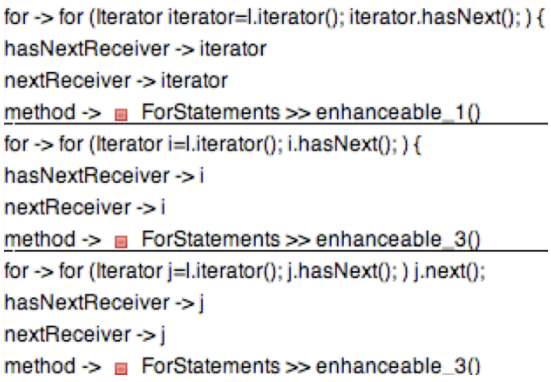


Figure 5.11: Results for Figure 5.6's query extended with ad-hoc data flow char. .

type shares the same limitations. Template terms are compiled to queries that use the traversal predicates of CAVA.

5.3.4 Expressing Data Flow Characteristics

The CAVA library does not provide predicates that reify the data flow analysis results used by the domain-specific unification procedure. Otherwise, users would have to quantify over these results in queries and interpret the solutions to these queries correctly. Both are problematic (cf. Section 2.5.4). Before illustrating this, we will show how LMP users often attempt to express data flow characteristics in terms of syntactic characteristics. Concretely, we will specify and detect (in an ad-hoc manner) the data flow characteristics of enhanceable `for`-statements and the protocol discussed in the previous section.

Expressing Data Flow Characteristics in Terms of Syntactic Characteristics

The query depicted in Figure 5.6 only expresses the syntactic characteristics of potentially enhanceable `for`-statements. Their data flow characteristics state that the receiver of the invocations `hasNext()` and `next()` should be the same iterator object. This can be expressed in terms of syntactic characteristics by adding the following conditions to the query:

```
1    ?hasNextReceiver simpleNameHasIdentifier: ?id,
2    ?nextReceiver simpleNameHasIdentifier: ?id
```

The additional conditions require both receivers to be `SimpleName` nodes with unifying identifier strings (e.g. `"iterator"` in method `enhanceable_1`). The query will not recognize `for`-statements in which the receivers of these invocations are other AST nodes.

Figure 5.11 depicts the solutions to the extended query against the program in Figure 5.6. The statement in method `not_enhanceable_1` is no longer recognized as enhanceable. This is correct. The statement in method `enhanceable_2` was reported by the original query, but is no longer included in the results to the extended query. The identifiers of the receivers of the `hasNext()` (`"i"`) and `next()` (`"j"`) invocations differ syntactically. The `for`-statement in method `enhanceable_4` is not included either. The receivers of the `hasNext()` (`((Iterator) temp)`) and `next()`

```
+?typeDeclaration declaresType: ?aType if
  [?typeDeclaration resolveBinding getJavaElement] equals: ?aType.
+?method1 overrides: +?method2 if
  ?method1 isMethodDeclaration,
  ?method2 isMethodDeclaration,
  [?method1 resolveBinding overrides_IMethodBinding: ?method2 resolveBinding]
```

Figure 5.12: CAVA's basic reasoning predicates rely on semantic analysis results.

(i) invocations are not SimpleName nodes. However, both statements can be enhanced. In this case, expressing the data flow characteristics in terms of syntactic characteristics resulted in pattern instances being missed.

Note that method enhanceable_3 features twice in the solutions to this query. The false positive in which *?nextReceiver* →j is not the iterator used in the outer for-statement *?for* →for(Iterator i = l.iterator(); i.hasNext();) is eliminated. The query uses the data flow characteristics of the pattern to eliminate this false positive rather than extra conditions on the nesting of *?nextReceiver* within one of the *?updaters* of the for-statement.

The data flow characteristics of methods that comply with the protocol described in Section 5.3.3 are even harder to express correctly in terms of syntactic characteristics. The protocol requires that the invocation of method c(Object) takes the result of a prior invocation of method a() as its argument. We could try to add the following condition to the query in the bottom-left corner of Figure 5.10:

```
  ?c methodInvocationHasArguments: ?args,
  [?args size = 1],
  ?args contains: ?a
```

According to the resulting query, only methods compliant_2 and compliant_5 in Figure 5.9 comply with the control flow and data flow characteristics of the protocol. In the control flow of these methods, invocation *?a* is the actual argument of invocation *?c*. The query fails to recognize method compliant_1, for instance, because it assigns the result returned by *?a* to a local variable that is used as the argument for *?c*. This is an implicit point of variation among complying methods. It could be specified as an explicit variation point in the query (e.g. using a disjunction or an alternative query). However, some implicit variation points can only be recognized by analyzing the entire program. This is, for instance, the case for method compliant_3 which invokes a method that returns the result of *?a*. Method compliant_4 assigns the result of *?a* to a field that is used as the argument of *?c*.

Evaluation Compared to the original query for the enhanceable for-statement, the extended query was able to eliminate a false positive by expressing the pattern's data flow characteristics —albeit in an ad-hoc manner. This illustrates the importance of these characteristics.

However, it is difficult to enumerate all implicit points of variation among the implementations of a data flow characteristic and express them in terms of syntactic characteristics. This is evidenced by the specification for methods that comply with the protocol. General-purpose pattern detection tools should therefore explicitly support data flow characteristics —as required by criterion **CSL1**.

Expressing Data Flow Characteristics Directly

The CAVA library does not support expressing data flow characteristics directly. The data flow analyses results in the program representation are not reified.

The implementation of some of the basic reasoning predicates uses the results of the semantic analysis internally. These predicates have to relate type declarations to the types they declare, implement or extend (cf. Section 5.2.2). The Eclipse JDT Core Component [Ecl08a] provides a convenient API to query its semantic analysis for the results for a specific AST node. Invoking method `resolveBinding()` on (among others) method declaration, type declaration, name and type nodes results in a "binding". This binding represents a fully qualified named entity in the program under investigation (i.e. an entry in the symbol table of the Eclipse compiler). Given an AST node, the implementation of most basic reasoning predicates merely has to consult its binding. This is illustrated by the rules depicted in Figure 5.12.[18] The first rule demonstrates that the binding for an AST node can be mapped back to an element in the structural program information. Note that the depicted predicates hide the details from the semantic analysis results used in their implementation. This is in compliance with criterion **CDM1**. Otherwise, details internal to the Eclipse compiler would pop up in solutions to queries. We will revisit the semantic analysis in the discussion of the domain-specific unification procedure (cf. Chapter 6).

The results of the context-insensitive points-to analysis computed by the SPARK [Lho02] component of the SOOT Java Optimization Framework [VRCG+99] are not reified either. There are no technical problems in the way. We could have reified the results through the linguistic symbiosis with Java. In fact, the points-to analysis results depicted in Figure 2.6 were obtained by backtracking over the first six conditions of the query in Figure 5.13. The analysis is computed for the JIMPLE intermediate representation rather than the AST nodes in our representation. The query finds all pairs of local variables *?local1* and *?local2* in this representation that are in a may-alias data flow relation. The latter is checked by the condition on line 12 which requires the points-to sets *?set1* and *?set2* for the locals to have a non-empty intersection.

Evaluation The predicates in Figure 5.13 form a superficial logic interface to the API of the SOOT framework. In this, they are reminiscent of the DEEPWEAVER [FKI+07] predicates depicted in Figure 3.16 and Figure 3.17 which rely on the same framework. *We present these predicates in the traditional Prolog notation to stress that they are not part of our instantiation of the LMP cornerstone.* The predicates exemplify all of the problems data flow information poses in a pattern detection setting (cf. Section 2.5.4):

- The points-to analysis results come overlaid on the JIMPLE intermediate representation. This is a typed three-address representation in which all instructions take the form of two operands, an operation and a result. It is constructed from bytecode. Bytecode instructions that manipulate the stack have been eliminated by introducing local registers for implicit stack locations. JIMPLE's grammar is compact compared to the amount of bytecode instructions. However, users are still burdened with a non-trivial program

[18] The rules that handle unbound variables *?typeDeclaration*, *?method1* and *?method2* are, in contrast, complicated.

representation if they want to query the points-to analysis results. Figure 2.4 depicts the JIMPLE representation of the method queried in Figure 5.13. It differs significantly from the concrete syntax of the method depicted in Figure 2.1.

- The query in Figure 5.13 enumerates all local variables in the intermediate representation. Establishing the mapping between arbitrary AST nodes depicted in Figure 2.2 and the intermediate representation depicted in Figure 2.4 is difficult and contributes to the operational nature of pattern specifications (cf. Section 2.5.1). This mapping is required for patterns that are characterized by both behavioral and non-behavioral characteristics.

- The points-to analysis queried in Figure 5.13 is context-insensitive. It does not use any context-sensitive parametrizations for its static representation of heap references (cf. Section 2.5.4). Its results can therefore be accessed without having to provide a static representation of the run-time context in which a reference resides (e.g. the call sites of the topmost invocations on the call stack). The elements in the points-to set for such a reference (i.e. static representations of heap referees) are not parametrized by contexts either. They can thus be used without having to interpret static representations of run-time contexts. However, such contexts must be accounted for when the more precise analyses of the framework are used.

- Read literally, solutions to the query in Figure 5.13 consist of pairs of local variables such that their associated points-to sets have a non-empty intersection. Correctly interpreted, this entails that the local variables are in a *may-alias* data flow relation. A *must-alias* relation cannot be derived from these results. To correctly assess the solutions to a query that quantifies over data flow analysis results, users need to know which data flow relations can be derived from the results. Moreover, solutions to the query will inevitably include local variables that are never in an alias relation at run-time. This is because of imprecision in the analysis. To assess the solutions to the query, users need to be aware of the trade-offs with respect to precision and cost implemented by the analysis they quantify over.

A predicate library that reifies the results of the analysis at a higher level of abstraction could alleviate some of these problems. Providing a predicate `mayAlias/2` that reifies the may-alias relation between AST nodes would solve the first two problems. However, users would still have to provide a static representation of a run-time context to access context-specific results. Moreover, providing a predicate library to specify data flow characteristics does not assist users in their assessment of the reported results. Therefore, data flow analysis results are not reified in our approach. They are incorporated in the domain-specific unification procedure instead. This way, users can benefit from these results without being exposed to their details.

5.4 Open Implementation

We introduced the open implementation of SOUL in Section 4.6. SOUL provides reflective predicates that form a meta-interface through which its proof procedure can be manipulated. However, detailed knowledge about the internals of SOUL is

```
1  if ?s equals: ['<examples.Example: void insertElement(java.lang.Object)>'],
2     sSignatureOfMethod(?s,?method),
3     sActiveBodyOfMethod(?body,?method),
4     sLocalsOfBody(?locals,?body),
5     member(?local1, ?locals),
6     equals(?set1, [Soul.MLI forJavaBytecode
7                    pointsToAnalysis reachingObjects_Local: ?local1]),
8     member(?local2, ?locals),
9     not(equals(?local1, ?local2)),
10    equals(?set2, [Soul.MLI forJavaBytecode
11                   pointsToAnalysis reachingObjects_Local: ?local2]),
12    [?set1 hasNonEmptyIntersection_PointsToSet: ?set2]
```

Figure 5.13: How *not* to quantify over the may-alias relation of local variables.

required to implement custom pattern search strategies in this manner. Implementing a meta-interpreter is therefore a better option (cf. Section 4.6.1).

The meta-interpreter in Figure 5.2 only uses reflective predicates to clarify the handling of Smalltalk terms. Lines 3–5 can be replaced by a single condition *&goal*. Users are thus not necessarily exposed to implementation details.

5.5 Limitations of the Instantiation

The instantiation of the LMP cornerstone discussed in this chapter, SOUL and the CAVA library, still have some technical limitations.

5.5.1 Performance Disadvantage of SOUL

Performance-wise, SOUL lags behind other Prolog implementations. Compared to the open source GNU PROLOG and SWI-PROLOG on the 10-queens problem, SOUL is about a factor of 5 and 20 slower respectively. The SOUL evaluator is an interpreter implemented in Smalltalk. SOUL programs are not compiled to Smalltalk byte codes. Its design facilitates exploring non-standard syntax and proof procedures that are suitable for logic meta programming. Few optimizations have been incorporated. This should be taken into account when assessing the running times of the queries in this dissertation.

To cope with this performance disadvantage, the CAVA library makes extensive use of mode annotations and caching. The former allow specializing the rules that are used for a predicate depending on its context of use (cf. Section 5.1.1). The latter applies to the results of predicates that are used often. Otherwise subsequent uses of predicate `isExpressionInScopeOf:/2`, for instance, would lead to identical AST traversals (cf. Section 5.3.3).

We also retrieve and cache the Java objects in the program representation ahead of time. Otherwise, the JAVACONNECT library would still have to create Smalltalk proxies for Java objects during the evaluation of queries.

In future work, we want to incorporate tabled resolution [RC97, CW96] in SOUL. This would obviate the need to manually cache the results of strategic predicates and allow left-recursion in the Definite Clause Grammar for the code in template terms (cf. Section 4.3.3). Linguistic symbiosis might provide a new implementation technique for incorporating tabling in an existing Prolog engine as well.

5.5.2 Technical Limitations of CAVA's Control Flow Traversal Predicate

The control flow traversal predicates in the CAVA library are limited in the control flow characteristics they support:

Inaccurate control flow graph The control flow graph in the program representation of our prototype is not very accurate. The effect of exceptions, for instance, is not taken into account. The graph has inter-procedural back-edges (e.g. for recursive methods), but does not have intra-procedural back-edges (e.g. for while loops). This is only a technical limitation of the prototype that computes control flow edges on-the-fly.

In future work, we want to adopt an accurate control flow graph. However, its nodes should still be nodes from the AST. This rules out using the control flow graphs from the Soot framework (cf. Figure 2.5).

No support for complements in existential path queries In essence, subsequent control flow traversals express an existential path query. Consider the conditions on lines 6–10 of the bottom-left query in Figure Figure 5.10. They correspond to the existential query: *"does there exist a path through* ?block *on which* ?c *follows* ?a *?"*. Negating a series of subsequent control flow traversals renders the query universal: *"is it true that there is not a single path . . . ?"*. It is not possible to express an existential path query with a complement: *"does there exist a path through* ?block *on which anything but* ?c *follows* ?a*?"*. We will illustrate this shortcoming of the CAVA library using the examples below.

The following query demands that all paths after *?a* contain instructions that are anything but *?c*:

```
1  if ...
2     ?a inFlowOf: ?block following: <> before: <>,
3     ?a methodInvocationHasName: simpleName(['a']),
4     not(and(?c inFlowOf: following: <?a> before: <>,
5             ?c methodInvocationHasName: simpleName(['c'])))
```

It has only methods `e` and `not_compliant_1` in Figure 5.9 as solutions. Method `compliant_4` is, for instance, not recognized even though it has a path on which *?a* is not followed by *?c*.

The following query, on the other hand, only filters out bindings for *?c* that are not invocations of a method named `c`. A binding for *?c* can even originate from a path on which an invocation of `c` follows the invocation of `a`.

```
1  if ...
2     ?a inFlowOf: ?block following: <> before: <>,
3     ?a methodInvocationHasName: simpleName(['a']),
4     ?c inFlowOf: following: <?a> before: <>,
5     not(?c methodInvocationHasName: simpleName(['c']))
```

The query only fails for methods `c`, `b` and `a` in Figure 5.9. They have no invocation of method `a` in their control flow graph. The query succeeds for all other methods.

Finally, the following query states that there is not a single path through *?block* on which *?a* is followed by *?c*:

```
1  if ...
2     not(and(?a inFlowOf: ?block following: <> before: <>,
3             ?a methodInvocationHasName: simpleName(['a']),
4             ?c inFlowOf: following: <?a> before: <>,
5             ?c methodInvocationHasName: simpleName(['c'])))
```

It has methods a,b,c,e and `not_compliant_1` as solutions. All paths through the latter method contain only a single method invocation.

The control flow interpretation of template terms (cf. Section 4.3.2) shares the same limitation. Under this interpretation, their source code excerpts are compiled to queries that use the traversal predicates of CAVA. Moreover, complements cannot be indicated in these excerpts without introducing more non-native syntax.

In future work, we want to change the translational semantics of the control flow interpretation to properly support universal and existential path queries with complements. Candidate algorithms to target are inter-procedural versions of the parametric regular path expressions proposed by Liu et al. [LRY+04] or the algorithm by de Moor et al. [dLW03] that is the basis for path logic programming [DdMS02] (cf. Section 3.4.3). Linguistic symbiosis facilitates integrating a straightforward Smalltalk implementation of the former algorithm. The latter algorithm requires tabled resolution. If SOUL is extended to support tabled resolution, a model checker for CTL formulas (computational tree logic [CES86]) over the control flow graph could also be incorporated in a straightforward manner (cf. for instance [RRR+97]).

5.6 Conclusion

In this chapter, we discussed the instantiation of the logic meta programming cornerstone. It consists of SOUL and the CAVA library for reasoning about Java programs.

We clarified how SOUL differs from regular Prolog through a meta-interpreter. Key features are its open implementation and its symbiosis with Smalltalk. The latter enables quantifying over any object that is reachable in the Smalltalk runtime image —including Smalltalk objects that function as proxies for Java objects. This is how we transitively established a symbiosis between SOUL and Java. We illustrated how we adapted the standard library predicates accordingly.

We discussed the predicates in the CAVA library. These can be used to quantify directly over the syntactic, structural and control flow information in our program representation. We do not provide reification predicates for the data flow analyses used by the domain-specific unification procedure. Otherwise, users would have to quantify over their results in queries and interpret the solutions to these queries correctly. Unique is its identity-based reification to objects: the reified version of an AST node is the AST node itself (i.e. an instance of `org.eclipse.jdt.core.dom.ASTNode`). It is enabled by the linguistic symbiosis with Java. This way, reconstructing the actual AST node from its reified counterpart is trivial at any point in the proof procedure (e.g. in the unification procedure). Moreover, the node's context within the program can be obtained through message sends.

We specified representative patterns for each characteristic as a logic query and identified the unification-related and quantification-related shortcomings of LMP as manifested in these specifications. Future chapters will therefore revisit these examples to show how these shortcomings are remedied by the other cornerstones of our approach. Only the patterns that are primarily characterized by structural characteristics need little improvement. The relational nature of logic programming facilitates quantifying over the reification predicates for structural information to express their structural characteristics.

CHAPTER 6

INSTANTIATING THE FUZZY LOGIC AND DOMAIN-SPECIFIC UNIFICATION CORNERSTONES

In this chapter, we discuss the instantiations of the fuzzy logic and domain-specific unification cornerstones. Both instantiations adapt the logic meta programming instance discussed in the previous chapter. The fuzzy variant of SOUL *initiates our discourse. It quantifies the truth of solutions to a logic query. This establishes a ranking among solutions which facilitates their assessment. We clarify the syntactic and semantic differences from regular* SOUL *using a meta-interpreter. The logic rules used in the resolution of a goal determine the upper bound for the truth degrees of its solutions. A solution can be ranked lower than the other solutions identified by the same rules. This is the case if it requires a domain-specific unification that could introduce false positives. Program analyses are used by the procedure to recognize implicit points of variation among pattern instances. False positives can stem from imprecision in these analyses. We will demonstrate how both cornerstones improve the support for pattern characteristics offered by logic meta programming.*

6.1 Fuzzy Variant of SOUL

We introduced and motivated the fuzzy logic cornerstone of our approach in Section 4.5. It can be incorporated in any pattern detection tool based on a machine-executable proof procedure for a logic formalism. Incorporated into SOUL (cf. Section 5.1), this cornerstone gives rise to a fuzzy logic programming language.

6.1.1 Syntax and Semantics in a Nutshell

Many "fuzzy Prolog" systems exist (cf. Section 4.5). The fuzzy variant of SOUL is close to F-PROLOG [LL90]. Its model, operational and fix-point semantics are de-

tailed in [De 04] where we applied an older incarnation to the detection of patterns through dynamic analysis.

We briefly recapitulate the core syntax and semantics of fuzzy SOUL from Section 4.5. Figure 4.11 depicts a fuzzy SOUL program. The fuzzy facts on lines 2–3 are annotated with truth degrees. This is also the case for the fuzzy rule on lines 7–9. In brief, fuzzy rules are of the following form:

$$q : c \ \textit{if}\ q_1, \ldots, q_n.$$

The rule has a head q and a body $q_1, \ldots, q_n$. It is annotated with a truth degree $c \in]0,1]$. This degree is interpreted as the confidence in a solution to goal q given the absolute truth of the sub-goals $q_1, \ldots, q_n$. It is the upper bound for the truth degrees of all solutions identified by this rule. Consider the fuzzy rule on lines 7–9 in Figure 4.11. The truth degrees for solutions to the query "- *if ?product* isPopular" identified by this rule cannot exceed 0.8. The truth degree of goal q is computed as the product of c and the minimum of the truth degrees of the sub-goals $q_1 \ldots q_n$. Hence, we quantify conjunction and implication as minimum and product respectively. This is the predominant quantification used by "fuzzy Prolog" systems (cf. Section 4.5). The truth degree computed for the solution *?product* → `chips` to the aforementioned query is therefore $\min(\frac{9}{10}, \frac{6}{10}) \cdot \frac{8}{10} = \frac{48}{100} = 0.48$.

Logic Variables as Annotations

Fuzzy SOUL supports two exceptions to the syntactic form of rules described above. Rules can be annotated with a logic variable rather than the real c. Eventually, this variable has to get bound in the body of the rule.

A goal can also be annotated with an unbound variable or a real (i.e. the annotated goal $q : t$). If t is an unbound variable, it will be bound to the truth degree of inner goal q. The truth extracting goal as a whole succeeds with a truth degree of 1 —neutralizing its influence. If t is a real, it serves as a threshold for the truth degree of inner goal q. The goal $q : 0.7$ succeeds with absolute truth (i.e. a truth degree of 1) if the truth degree of q is equal to or greater than 0.7. The goal fails otherwise. Solutions to the following query therefore consist of bindings *?t2* → 1 and *?t1* → t where t is the truth degree of the inner goal q:

```
1  if (q : ?t1) : ?t2
```

Should the need arise, such truth extracting goals allow overriding the default quantification of logic connectives —although only in an operational manner. For instance, to implement linguistic hedges such as *fairly* and *more or less*. In fuzzy set theory, these modify the membership degrees of the elements in a fuzzy set (cf. Kerre et al. [KC99] for an overview). The following higher-order rule could, for instance, be used to implement the linguistic hedge *very*:

```
1  (?goal) very : ?implicationStrength  if
2          ?goal : ?truth,
3          [?truth squared] equals: ?implicationStrength
```

The truth degree computed for the solution *?product* → `chips` to the query "*if* `(?product isPopular) very`" is $(\frac{48}{100})^2 = 0.23$

6.1.2 Meta-Interpreter for Fuzzy SOUL

The meta-interpreter depicted in Figure 6.1 further clarifies the semantics of the fuzzy variant of SOUL. We will concentrate on the differences from regular SOUL.

Compared to the meta-interpreter for regular SOUL (cf. Section 5.1.2), its clauses take two extra arguments *?degree* and *?threshold*. The former represents the extent to which the truth of *?goal* can be proven. The latter represents a threshold for this truth degree. The following query, for instance, computes the truth degree of the goal `[9/10]`:

```
1  if [Soul.Maybe degree: 9/10] isProvenToExtent: ?degree aboveThreshold: 0
```

It will result in a binding *?degree* → $\frac{9}{10}$.

Handling of Smalltalk Terms

Fuzzy SOUL and regular SOUL differ in how they handle Smalltalk terms that are used as goals. Rather than requiring the expression in the term to evaluate to the boolean `true`, fuzzy SOUL requires it to evaluate to an object that understands the message `degree` (line 5 of Figure 5.2 versus line 5 of Figure 6.1). The instances of singletons `True` and `False` respond to this message with 1 and 0 respectively. Instances of class `Maybe` respond with the particular truth degree they wrap (0.9 for the instance created in the query above). The resolution of the Smalltalk term is quantified by the truth degree *?degree* of its expression (first condition on line 5).

Handling of Annotated Goals

In general, a goal succeeds if its truth degree lies above 0 (second condition on line 5) and satisfies the threshold *?threshold* (third condition on line 5). Otherwise, the goal fails.

Lines 20–23 and lines 24–26 handle goals that are annotated with a real and an unbound variable respectively. Line 23 illustrates that the former annotations impose a threshold for the truth degree of the inner goal. Line 26 illustrates that the latter annotations unify with the truth degree of the inner goal. The complete goals succeed with a truth degree of 1 if their inner goals succeed. Thresholds for the truth degrees of the complete goals therefore need not be checked (*?* on lines 20 and 24) and are not imposed on the inner goal (line 26).

Quantification of Logic Connectives

Lines 6–19 clarify how the variable-argument connectives `and/n`, `or/n` and `not/n` of SOUL are quantified. The truth degree of a goal `and@(`*`?goals`*`)` is the minimum of the truth degrees of its argument goals. We will discuss the predicate used on lines 8–9 to compute this degree in the section on quantified resolution.

A solution to a goal `or@(`*`?goals`*`)` is identified by one or more of its argument goals (lines 12–13). Multiple truth degrees can therefore be associated with the same solution (i.e. variable bindings). The fuzzy SOUL prototype does not aggregate identical solutions with different truth degrees into a single solution with an aggregated truth degree (e.g. the maximum of these degrees as computed by van Emden's quantitative rules [vE86] or Li's F-PROLOG [LL90]).[1] This exposes the

[1] For consistency reasons, it does not aggregate over solutions to a goal that can be resolved using multiple fuzzy rules either.

```
&goal isProvenToExtent: ?degree aboveThreshold: ?threshold if
  [&goal isKindOf: Soul.SmalltalkTerm],!,
  getEnv(?env,?),envLookup(&goal,?gpointer),
  ?value equals: [?gpointer term evaluateIn: ?env startAt: ?gpointer envIndex],
  [?value degree] equals: ?degree,[?degree > 0],[?degree >= ?threshold].

?goal isProvenToExtent: ?degree aboveThreshold: ?threshold if
  ?goal equals: and@(?goals),!,
  ?goals isProvenListOfGoalsToExtent: ?degree aboveThreshold: ?threshold
         runningMin: [1] implicationStrength: [1].
?goal isProvenToExtent: ?degree aboveThreshold: ?threshold if
  ?goal equals: or@(<&h|?t>),not(?t equals: <>),!,
  or(&h isProvenToExtent: ?degree aboveThreshold: ?threshold,
     or@(?t) isProvenToExtent: ?degree aboveThreshold: ?threshold)
?goal isProvenToExtent: ?degree aboveThreshold: ?threshold if
  ?goal equals: or(&h),!,
  &h isProvenToExtent: ?degree aboveThreshold: ?threshold
?goal isProvenToExtent: ?degree aboveThreshold: ?threshold if
  ?goal equals: not@(?goals),!,
  ?goal : ?degree,[?degree >= ?threshold]

?goal isProvenToExtent: [1] aboveThreshold: ? if
  equals(?goal,&innergoal : ?annotation),nonvar(?annotation),!,
  [?annotation isReal],
  &innergoal isProvenToExtent: ? aboveThreshold: ?annotation.
?goal isProvenToExtent: [1] aboveThreshold: ? if
  equals(?goal,&innergoal : ?annotation),var(?annotation),!,
  &innergoal isProvenToExtent: ?annotation aboveThreshold: [0].

?goal isProvenToExtent: ?degree aboveThreshold: ?threshold if
  ?goal isHeadOfRuleWithBody: ?conditions andImplicationStrength: ?i,
  ?conditions isProvenListOfGoalsToExtent: ?degree aboveThreshold: ?threshold
              runningMin: [1] implicationStrength: ?i

<> isProvenListOfGoalsToExtent: ?implication aboveThreshold: ?threshold
   runningMin: ? implicationStrength: ?implication if
  [?implication >= ?threshold].

<&last> isProvenListOfGoalsToExtent: ?degree aboveThreshold: ?threshold
        runningMin: ?currentMin implicationStrength: &implication if
  !,&last isProvenToExtent: ?d aboveThreshold: ?threshold,
  ?min equals: [?currentMin min: ?d],
  ?degree equals: [?min * ?implication],
  [?degree >= ?threshold].

<&g|&r> isProvenListOfGoalsToExtent: ?degree aboveThreshold: ?threshold
        runningMin: ?currentMin implicationStrength: if
  &g isProvenToExtent: ?d aboveThreshold: ?threshold,
  ?min equals: [?d min: ?currentMin],
  &r isProvenListOfGoalsToExtent: ?degree aboveThreshold: ?threshold
     runningMin: ?min implicationStrength: ?implication
```

Figure 6.1: The meta-interpreter corresponding to the fuzzy variant of SOUL.

operational nature of its goal-oriented proof procedure, but clarifies the origin of each pattern instance (i.e. solution) presented in this dissertation. For instance, whether an instance was identified by multiple example-based interpretations of the same template term. If required, users can still perform an explicit aggregation step manually (e.g. using the higher-order `findall/3` predicate).

The meta-interpreter in Figure 6.1 does not clarify how a goal `not@(?goals)` is quantified. The underlying interpreter computes the truth degree and the meta-interpreter only verifies that the threshold is met (line 19). The meta-interpreter should therefore be evaluated in fuzzy SOUL (i.e. it is meta-circular). The fuzzy `not/n` connective succeeds as long as the conjunction of its arguments does not succeed completely (i.e. truth degree < 1). If the conjunction of its arguments fails, the fuzzy `not/n` connective succeeds with the complete truth degree of 1. If the conjunction of its arguments succeeds with a truth degree d, the connective succeeds with a truth degree $1 - d$. The prototype therefore quantifies negation as failure using complement (i.e. the original De Morgan style quantification defined by Zadeh [Zad65]).

Quantified Resolution

Lines 27–45 illustrate quantified resolution. Line 28 looks up a rule of which the head unifies with the current *?goal*. Its *?conditions* are used as a new list of goals to resolve. Their resolution is handled by predicate `isProvenListOfGoalsToExtent:aboveThreshold:runningMin:implicationStrength:/5`. If the rule is annotated with a truth degree (i.e. a fuzzy rule), it is passed as the last argument to the predicate. If the rule is not annotated with an explicit truth degree (i.e. a crisp rule), it has an implicit truth degree of 1.

The rules that implement the predicate (lines 31–45) keep a running minimum that represents the smallest truth degree among the goals evaluated so far. The third rule is the recursive rule. It updates the running minimum on line 43. The second rule handles the last goal in the list. On line 38, it multiplies the smallest truth degree among the goals (*?min*) with the truth degree the rule is annotated with (*?implication*). Line 39 verifies whether or not the threshold is met. [2]

The quantified resolution procedure (described here) is incorporated into SOUL by specializing the object-oriented implementation of the regular resolution procedure. Some essential ingredients of this procedure are not made explicit by the meta-interpreter. For instance, the procedure considers candidate rules for the resolution of a goal in the order in which they are defined in the program. Rules with a higher truth degree are not considered first.

Combining Resolution Degrees with Unification Degrees

The meta-interpreter does not clarify how unification degrees can lower the truth degrees of solutions either (cf. Section 4.5). The rule in Figure 6.2 clarifies

[2] The truth degree of a fuzzy rule is an upper bound for the truth degrees of the solutions it identifies. On line 28, this can be used to discard rules that will not lead to solutions that meet the imposed threshold. It can also be used on line 42 to stop evaluating the remainder of goals if the truth degree of the current goal precludes the threshold from being met (i.e. using a condition `[(?d * ?implication) >= ?threshold]`). However, allowing rules to be annotated with variables precludes incorporating this optimization in a straightforward manner. Such a variable may not receive its binding until the very last goal in the list.

```
1  ?goal isProvenToExtent: ?degree aboveThreshold: ?threshold if
2    &head isHeadOfRule: ?rule withBody: ?conditions andImplicationStrength: ?i,
3    getEnv(?env,?),envLookup(&head,?headp),envLookup(?goal,?goalp),
4    [?env startUnifyWith: ?rule],
5    ?udegree equals: [((?headp term) unifyWith: (?goalp term)
6                                     inEnv: ?env
7                                     myIndex: (?headp envIndex)
8                                     hisIndex: (?goalp envIndex)
9                                     inSource: true) degree],
10   [(?udegree > 0) ifTrue: [true] ifFalse: [?env rollback. false]],
11   ?conditions isProvenListOfGoalsToExtent: ?rdegree
12               aboveThreshold: ?threshold runningMin: 1 implicationStrength: ?i,
13   ?degree equals: [?rdegree * ?udegree],
14   [?degree >= ?threshold]
```

Figure 6.2: Meta-interpreter excerpt clarifying handling of unification degrees.

how unification degrees are handled. The rule can substitute for the rule on lines 27–30 of the meta-interpreter.[3]

The truth degree *?rdegree* obtained by resolving *?goal* using the *?conditions* of a *?rule* (lines 11–12) is multiplied with the extent *?udegree* to which the head of the rule and the goal unify (line 13). Unification degrees quantify the extent to which two terms unify. Two terms do not unify if the computed unification degree is 0. In this case, the rule cannot be used to resolve the goal and changes to the environment *?env* are rolled back (line 10). The Smalltalk term evaluates to `false` to ensure that the first condition is backtracked over to consider other candidate rules.

We use multiplication rather than minimum to compute the truth degree from the resolution and unification degrees. This way, both influence the truth degree for the goal.[4] Section 6.8 elaborates on the combination of unification and resolution degrees.

Method `unifyWith:inEnv:myIndex:hisIndex:inSource:` implements the unification procedure. It is invoked on object *?headp* which implements the term that represents the head of the rule (lines 5–9). This receiver and the other arguments of the invocation are retrieved using the reflective predicates of SOUL (Section 5.1.2 discusses their use in the vanilla meta-interpreter). The invocation returns either an instance of the singletons `True` and `False` or an instance of class `Maybe` containing a unification degree between 0 and 1.

Composite Unification Degrees

In our approach, unification degrees only arise when unifying two reified program elements could introduce false positives (cf. Section 6.4). However, reified program elements can be used in compound terms, lists and predicates (i.e. composite terms). The unification degree of two composite terms is therefore computed by multiplying the unification degrees of their corresponding sub-terms. The more imprecise unifications of sub-terms are required, the lower the unification degree of the composites.

[3] Their behavior is identical if the fuzzy variant of SOUL is used to evaluate the meta-interpreter in Figure 6.1.

[4] Otherwise, all solutions to a template term that require a unification based on points-to analysis would for instance have the same truth degree regardless of whether or not the example-based interpretation itself could introduce false positives.

This is illustrated by the solutions to the following query that originate from method `indirectReturn(Object,int)` (cf. Figure 4.8):

```
1 if ?m methodDeclarationHasName: simpleName(['indirectReturn']),
2    ?e1 isExpressionInScopeOf: ?m, ?e2 isExpressionInScopeOf: ?m,
3    f(a,?e2,?e1) equals: f(a,?e1,?e2) : ?t
```

These include solution <*?t* → 1,*?e1* → o,*?e2* → o> and solution <*?t* → $\frac{1}{4}$,*?e1* → o,*?e2* → `indirectReturn(o,delay-1)`>. The last goal is implemented by the logic fact "*?x* `equals:` *?x*". The truth degree *?t* of the goal is therefore 1 (the implicit truth degree of the fact) multiplied by the unification degree of compound terms `f(a,`*?e2*`,`*?e1*`)` and `f(a,`*?e1*`,`*?e2*`)`. The unification degree of these composite terms is computed by multiplying the unification degrees of their corresponding sub-terms. In the first solution, *?e1* and *?e2* are bound to the same AST node. The unification degree of the compound terms is 1 because the unification degrees of their corresponding arguments are respectively 1, 1 and 1. In the second solution, *?e1* and *?e2* are bound to expressions that are in a may-alias relation. The unification degree of the compound terms is $\frac{1}{4}$ because the unification degrees of their corresponding arguments are respectively 1, $\frac{1}{2}$ and $\frac{1}{2}$ (cf. Section 6.4).[5]

6.2 Fuzzified Standard Library

The fuzzy version of SOUL extends and adapts the standard library of SOUL as well. We will discuss the predicates that are used by the queries in the remainder of this dissertation.

6.2.1 Support for Fuzzy Sets

Zadeh proposed fuzzy sets [Zad65] to capture vague concepts without sharp boundaries in human-like descriptions of systems. Examples include the fuzzy set of people that are tall or the fuzzy set of numbers that are close to 20. The fuzzy set A differs from crisp sets in that its characteristic function μ_A takes on values in the real interval $[0,1]$ for every element in its universe —membership degrees. The value $\mu_A(x)$ represents the membership degree of the element x in the fuzzy set A. Fuzzy logic has its roots in fuzzy set theory. The value $\mu_A(x)$ can also be seen as the truth degree of the proposition that states $x \in A$ is true.

Fuzzy sets are useful in specifications of patterns with vague classification boundaries (cf. Section 6.3). Examples include patterns describing bad smells [FBB+99] such as methods that are too long and patterns involved in the calculation of software metrics [LM06]. In fuzzy SOUL, users can define fuzzy rules to implement a predicate that reifies the characteristic function of a fuzzy set. Alternatively, the `contains:/2` predicate of its standard library can be used to quantify over the elements of a fuzzy set. Such sets are Smalltalk objects and can be instantiated with user-defined characteristic functions.

[5] One could argue that this solution to the query is ranked lower than required because unifying *?e1* and *?e2* should not introduce more false positives if *?e2* and *?e1* already unify. This is an issue of the implementation (which performs a recursive descent through the terms), not of our use of multiplication. If minimum were used, a composite of which 3 sub-terms are in a may-alias relation would not have a lower unification degree than one in which 2 sub-terms are in a may-alias relation. Medina et al. [MOAV04] and Gilbert et al. [GS00] moreover argue independently for the use of multiplication rather than minimum in unification degrees.

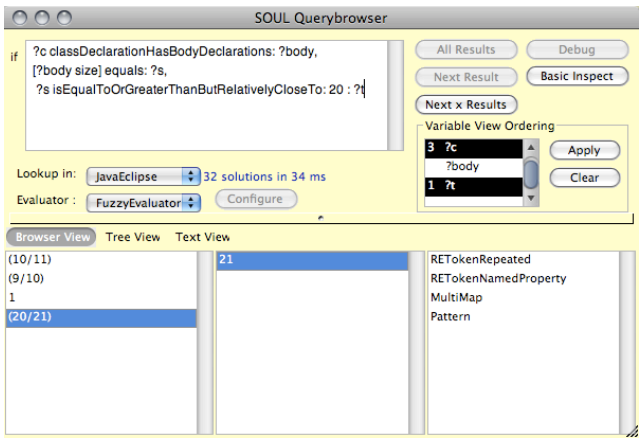


Figure 6.3: Illustrating fuzzy `isEqualToOrGreaterThanButRelativelyCloseTo:/2`.

We will demonstrate both techniques. The first technique illustrates that fuzzy SOUL supports rules annotated with a variable that gets bound in their body. The second technique illustrates that fuzzy SOUL supports Smalltalk terms with an expression that evaluates to a truth degree rather than a boolean.

1/ Implementing Predicates that Reify the Characteristic Function of a Fuzzy Set

Predicate `+?x isEqualToOrGreaterThanButRelativelyCloseTo:+?y` reifies the characteristic function of the fuzzy set of numbers that are greater than *?y*, but still relatively close to *?y*. Both arguments have to be bound. The following rules implement the predicate:

```
1  +?x isEqualToOrGreaterThanButRelativelyCloseTo: +?x.
2  +?x isEqualToOrGreaterThanButRelativelyCloseTo: +?y : ?c if
3    [?x > ?y],
4    ?c equals: [(?y / ?x) max: (9 / 10)]
```

Note that the second rule is annotated with a variable that gets bound in the body of the rule (cf. Section 6.1.1). It associates a truth degree $\in [\frac{9}{10}, 1[$ with numbers *?x* that are greater than *?y*, but do not deviate more than 10% from *?y*. The closer *?x* is to *?y*, the higher the computed truth degree. We do not let the truth degree drop below $\frac{9}{10}$ for numbers that lie far from *?y*.[6]

The first column in Figure 6.3 depicts the truth degrees for solutions to a query that uses the predicate to identify class declarations in the AMBIENTTALK interpreter (cf. Section 5.3.2) with more than 20 members. Except for the classes with 21 and 22 members (truth degrees $\frac{20}{21}$ and $\frac{10}{11}$ respectively), all classes with more than 20 members have a truth degree of $\frac{9}{10}$.

[6] The predicate is used in the resolution of template terms where it ensures that lower truth degrees are associated with solutions that exhibit more characteristics than the ones that are exemplified by a template (cf. Section 4.5.2). It is, for instance, used to compare the number of modifiers in the template (*?y*) to the modifiers in a solution (*?x*). Whether a reported method has more modifiers than specified should not affects its likelihood of being a false positive too much.

2/ Quantifying over Fuzzy Sets implemented in Smalltalk

Predicate `contains:/2` quantifies over the elements of a fuzzy set through linguistic symbiosis. Instances of class `FuzzySet` respond to message `membershipDegreeOfElement:` with the extent to which the argument can be considered an element of the set. The following rule implements the case in which both arguments of the predicate are bound:

```
1 +?c contains: +?e if
2   [?c isKindOf: Soul.FuzzySet],
3   [?c membershipDegreeOfElement: ?e]
```

Its implementation illustrates that Smalltalk terms are allowed to evaluate to a truth degree (cf. Section 6.1.2). Solutions to the following query include bindings $<?t \rightarrow 1, ?e \rightarrow 20>$, $<?t \rightarrow \frac{9}{10}, ?e \rightarrow 21>$ and $<?t \rightarrow \frac{4}{5}, ?e \rightarrow 22>$:

```
1 if ?about20 equals: [Soul.FuzzySet triangularWithPeak: 20 andMin: 10 andMax: 30],
2    [8 to: 32] contains: ?e,
3    ?about20 contains: ?e : ?t
```

The first line of the query instantiates a fuzzy set of which the elements are close to the number 20. Its triangular membership function $\Delta(x,10,20,30)$ determines the membership degree of element x. It linearly models how close x is to β ($\alpha < \beta < \gamma$):

$$\Delta(x,\alpha,\beta,\gamma) = \begin{cases} 0 & x < \alpha \\ (x-\alpha)/(\beta-\alpha) & \alpha \leq x \leq \beta \\ (\gamma-x)/(\gamma-\beta) & \beta \leq x \leq \gamma \\ 0 & x > \gamma \end{cases}$$

The membership function of a fuzzy set can also be instantiated with a custom `BlockClosure` or by enumerating its elements and their membership degrees.

6.2.2 Classical Negation as Failure

Unlike the regular connective, the fuzzy `not/n` connective (cf. Section 6.1.2) introduces choice points if the conjunction of its arguments can be proven to different extents. However, like the regular connective, variable bindings established by resolving this conjunction are undone.

Where choice points are undesirable, predicate `absolutelyNot/n` can be used as an alias for the regular `not/n` connective. It succeeds only if the fuzzy `not/n` connective succeeds with an absolute truth degree (i.e. if the conjunction of its arguments fails). The predicate is implemented as follows:

```
1 absolutelyNot@(?goals) if
2   not@(?goals) : 1
```

6.2.3 Higher-Order Predicates

The implementation of some higher-order standard library predicates is changed as well. Like the regular `forall/2` predicate, the fuzzy version of the predicate fails when there is a solution to the first argument goal for which the second argument goal does not succeed. Both versions differ in their quantification.

The fuzzy version of the predicate is quantified by the smallest product of truth degrees for each solution to its first argument and the corresponding solution to its second argument. Its implementation relies on linguistic symbiosis:

```
1 forall(?query,&test) : ?t if
2   ?degrees equals: [OrderedCollection new],
3   not(?query : ?queryTruth,
4       not(&test : ?testTruth,
5           [?degrees add: (?queryTruth * ?testTruth). true])),
6   ?t equals: [?degrees inject: 1 into: [:min :truth | min min: truth]]
```

The collection instantiated on line 2 contains the truth degree *?queryTruth* of each solution to goal *?query* multiplied by the truth degree *?testTruth* of the corresponding solution to goal *&test* (line 5).[7] Line 6 binds the truth degree of the rule to the minimum of these degrees. This is also the truth degree of `forall/2` goals because the goals in the body of the rule have a truth degree of 1.

Other quantifications are possible. We use minimum on line 6 because we regard a successful `forall/2` goal as a sequence of conjunctions. This is in line with our quantification of conjunction. In contrast, we use product on line 5 because neither *?query* nor *&test* subsumes the other goal. This way, the computed truth degree reflects the truth of both goals.

The bindings <*?about20* → `a FuzzySet`, *?t* → $\frac{7}{10}$> are the solution to the following query:

```
1 if ?about20 equals: [Soul.FuzzySet triangularWithPeak: 20 andMin: 10 andMax: 30],
2    forall([17 to: 23] contains: ?e, ?about20 contains: ?e) : ?t
```

The truth degree bound to *?t* is the smallest membership degree in the fuzzy set *?about20* of the elements in the interval [17,23]. It corresponds to the boundaries of the interval.

In case *?query* and *&test* are required to succeed completely (i.e. not with a truth degree < 1), the goal `forall(?query:1,&test:1)` as well as the annotated goal `forall(?query,&test):1` can be used. The former goal fails earlier.

This concludes our discussion of the standard library for fuzzy SOUL. The differences from the one for regular SOUL are straightforward. Moreover, with the exception of predicate `absolutelyNot/n`, most of the predicates presented here are not often used in pattern specifications. Instead, they are used in the resolution of template terms to rank their matches.

6.3 Logic Meta Programming with Fuzzy Logic

The fuzzy logic cornerstone enables our detection mechanism to quantify the pattern instances it reports with the extent to which they exhibit the characteristics expressed in a specification (cf. Section 4.5.2). The lower this extent, the more likely a result is a false positive. False positives may originate from an unintended example-based interpretation of the source code in a template term (cf. Section 4.3.2) or from imprecision in the static analyses that enable recognizing implicit points of variation among pattern instances (cf. Section 4.4). Facilitating user assessment of such results is the primary motivation for the fuzzy logic cornerstone.

Nonetheless, fuzzy logic meta programming has interesting applications of its own. We will briefly demonstrate two applications in which solutions are quantified by other information than their likelihood of being false positives.

[7] Variable *&test* is preceded by an ampersand in the head of the rule to delay the evaluation of the expression within Smalltalk terms to line 4 (cf. Section 5.1.2).

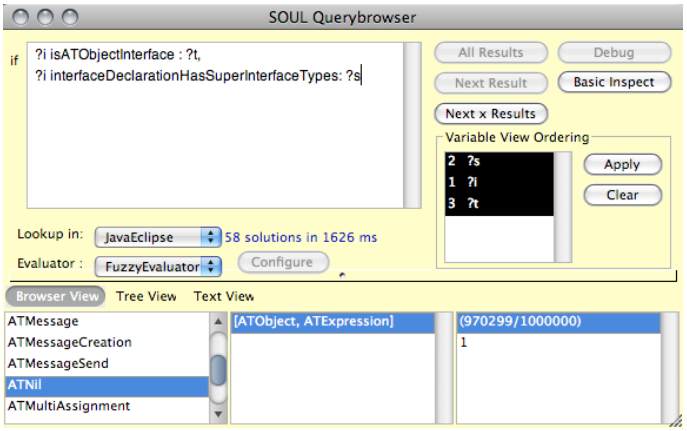


Figure 6.4: Interfaces in a hierarchy quantified by how close they are to the root.

Example 1: Quantifying the Distance to the Root of a Type Hierarchy

Figure 6.4 depicts the solutions to a query that identifies all interfaces that extend the `ATObject` interface of the AMBIENTTALK [Amb] interpreter. Section 5.3.2 introduced the application-specific predicate `isATObjectInterface/1` which reifies the `ATObject` interface hierarchy. Its implementation is depicted in Figure 5.7. It relies on predicate `interfaceExtends:/2` which is quantified similarly to predicate `extends:/2` depicted in Figure 4.13.[8]

In solutions to the query, the maximum truth degree 1 is associated with interfaces that extend the `ATObject` root interface itself. Interfaces that extend `ATObject` indirectly are identified with a lower truth degree of $(\frac{99}{100})^i$ where i is the depth in the hierarchy at which its direct super interface is found. Interface `ATNil` (selected in the first column of Figure 6.4) extends interface `ATObject` directly as well as indirectly through interface `ATExpression` (second column). It is therefore identified once with a truth degree of 1 and once with a truth degree of $(\frac{99}{100})^3$ because `ATExpression` resides at depth 3 in the interface hierarchy (third column).

The computed truth degrees clarify how far an interface lies from the root of the hierarchy. Substituting a truth degree for *?t* in the query, the interfaces can moreover be restricted to those that reside at a certain depth in the hierarchy.

Example 2: Detecting Patterns With Vague Classification Boundaries

Consider the bad smell [FBB+99] describing classes with many public static fields of a primitive type. These are often used to simulate C-like enumeration types in Java. In case classes with 10 fields are deemed a bad smell, it it is likely that a class with 9 fields is considered a bad smell as well. In regular LMP specifications, such vague classification boundaries have to be accounted for explicitly. For instance, by relaxing the classification boundary of the bad smell to 7 fields. In solutions to the resulting query, classes with 7 fields cannot be discerned from classes with 10 fields —although the former methods are less "bad".

Fuzzy SOUL supports fuzzy sets in specifications for patterns with vague classification boundaries (e.g. bad smells and patterns involved in the calculation of

[8] The predicates differ because interfaces can extend multiple super interfaces.

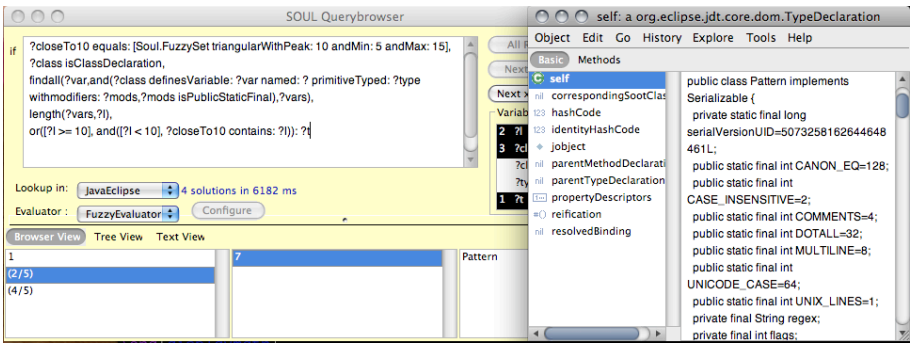


Figure 6.5: Quantified instances of the "many primitive public static final fields" bad smell.

software metrics [LM06]). Figure 6.5 depicts the quantified solutions to a fuzzy query that identifies the aforementioned bad smell in the AMBIENTTALK [Amb] interpreter. The query uses the fuzzy set *?closeTo10* of numbers that are close to 10 (first condition) to relax the classification boundary "more than 10 fields" to "more than 10 or close to 10 fields" (last condition). The query identifies, for instance, class `Pattern` as an instance of the bad smell with a truth degree of $\frac{2}{5}$. This is the membership degree of 7 (its number of fields) in the fuzzy set *?closeTo10* with characteristics function $\Delta(x,5,10,15)$ (cf. Section 6.2.1).

Like the regular LMP query, the fuzzy LMP query explicitly relaxes the classification boundary of the bad smell. Solutions to the fuzzy LMP query are, however, quantified by the membership degree of the number in the fuzzy set. This way, classes with 7 fields can be discerned easily from classes with 10 fields. Moreover, higher-order predicates that implement linguistic hedges such as *very* (cf. Section 6.1.1) can be used to highlight the worst instances of the bad smell (i.e. very long methods) in a descriptive manner.

In both applications of fuzzy LMP demonstrated above, the specification explicitly relaxes the classification boundary of a pattern to include imperfect pattern instances. In the first application, predicate `interfaceExtends:/2` relaxes the direct inheritance relation between an interface and its super interface to all ancestors of the interface. In the second application, fuzzy set *?closeTo10* relaxes the classification boundary of having more than 10 public static fields of a primitive type. There are no false positives: all reported instances adhere to the relaxed specification and the detection mechanism does not consult imprecise static analyses to recognize implicit points of variation among instances. Our detection mechanism does not automatically relax specifications to recognize imperfect pattern instances in this way. This would lead to many false positives.

6.4 Domain-Specific Unification Procedure for Java

Section 4.4 introduced and motivated the domain-specific unification cornerstone of our approach. This cornerstone overcomes the unification-related shortcomings of LMP in pattern detection. Terms that do not unify under the general-purpose procedure, can unify under the domain-specific procedure. The procedure computes a unification degree that reflects the likelihood that such a unification intro-

duces false positives.

Section 6.4.1 discusses domain-specific extensions to the general-purpose unification procedure that concern unifying two reified AST nodes. To recognize implicit points of variation among pattern instances, reified AST nodes unify if they represent different implementations of the same pattern characteristic. The results of whole-program analyses determine if this is the case for individual nodes. For instance, by unifying an unqualified type (e.g. `List`) with the qualified type it denotes according to the import declaration of the unit in which it resides (e.g. `java.util.List` with import declaration `java.util.List;` or `java.util.*`). Ad-hoc implementations of such comparisons lead to operational queries with false positives and a low recall (argued in Section 2.5 and illustrated in Section 5.3). Relying on the domain-specific unification procedure, users benefit from the results of its enabling analyses without being exposed to their details.

Section 6.4.2 discusses the extensions that concern unifying a reified AST node and other logic terms. Reified AST nodes unify with structurally equivalent compound terms to enable the natural use of unification to quantify over AST nodes (i.e. selecting nodes with specific children and accessing child nodes) —even if the reified version of an AST node is not a compound term. To support conditions over the source code that corresponds to an AST node, reified AST nodes also unify with terms representing a regular expression that matches their source code.

6.4.1 Unifying Reified AST Nodes

Whether two logic terms unify is determined by sending message `unifyWith:inEnv:myIndex:hisIndex:inSource:` to the Smalltalk implementation of one of the terms with the Smalltalk implementation of the other term as argument. This was illustrated by lines 5–9 of the meta-interpreter excerpt in Figure 6.2. SOUL already employs a modest extension to the regular unification procedure to accommodate unifying reified Smalltalk objects. Their implementation of `unifyWith:inEnv:myIndex:hisIndex:inSource:` invokes = on the corresponding Smalltalk objects. On instances of `org.eclipse.jdt.core.dom.ASTNode`, this method tests for object identity (i.e. ==). As a result, the general-purpose unification procedure only unifies identical AST nodes. Until our publications on the subject [DBD06, BDM07], the openness of SOUL (Smalltalk *Open Unification* Language) was never exploited to specialize the unification of AST nodes.

This section discusses our domain-specific extensions that concern how two AST nodes are unified. The domain-specific unifications are only attempted when the general-purpose unification (which amounts to an object identity test) fails. The most restrictive unification is attempted first and only upon its failure, a more relaxed unification is attempted. This ensures that two AST nodes always unify with the maximum unification degree. The unification procedure therefore does not introduce choice points in the refutation process that can be backtracked over.

According to the domain-specific unification procedure of our prototype, the following AST nodes unify. Their unification degree is 1 unless mentioned otherwise:

Unifying AST nodes are identical instances of `ASTNode` (i.e. the root class of the AST hierarchy). This ensures that the relation of unifying AST nodes is *reflexive* (i.e. xRx).

Unifying collections of AST nodes are `ASTNode$NodeList` instances of the same size for which all corresponding elements unify. Such collections contain the arguments of method invocations, the modifiers of method declarations, the statements in a block statement, etc. The computed unification degree is the product of the unification degrees of their corresponding elements. This is in line with the unification degree for two composite terms (cf. Section 6.1.2). The following query therefore always succeeds if *?collection1* and *?collection2* are unifying collections of AST nodes:

```
1  if ...
2     ?collection1 equals: ?collection2 : ?degree,
3     ?collection1 equals: nodeList(<?e1,?e2>)
4     ?collection2 equals: nodeList(<?e1,?e2>) : ?degree,
```

In the last condition, *?collection2* unifies to an extent of *?degree* with a structurally equivalent compound term. This is the extent to which both collections unify in the first condition because the compound term contains the elements of *?collection1*.

Unifying modifiers are instances of `Modifier` that represent the same modifier keyword (e.g. `abstract`). Instances that represent the same keyword would not unify according to the general-purpose procedure because they are different AST nodes.

Unifying formal parameters are `SingleVariableDeclaration` instances in the formal parameter list of a method declaration of which the types unify. The names of the parameters are not required to unify.

Unifying expressions are instances of `Expression` representing expressions of a reference type (i.e. a sub-type of `java.lang.Object` or an array of such types) that are in a may-alias relation according to the inter-procedural, context-insensitive points-to analysis (cf. Section 2.5.4) in our program representation (cf. Figure 4.1). The points-to sets of such expressions have a non-empty intersection. The procedure does not unify expressions with disjoint points-to sets. Such expressions can never evaluate to the same object because a points-to set includes all objects (i.e. static representations thereof) an expression can evaluate to.

The unification degree for expressions that are in a may-alias relation is $\frac{1}{2}$. Unifying such expressions can introduce false positives if the expressions do not alias during all possible program executions. Expressions in the same method that are in a may-alias relation unify with a higher unification degree of $\frac{9}{10}$ if they are also in a must-alias relation according to the intra-procedural must-alias analysis. Such expressions are guaranteed to alias in all possible program executions and can therefore never introduce false positives.[9]

Local expressions that are not in a may-alias relation can never be in a must-alias relation. The unification procedure fails early for such expressions. Expressions that are not in a must-alias relation (according to the intra-procedural must-alias analysis) can still be in a may-alias relation (accord-

[9] A unification degree of 9/10 rather than 1 accounts for unlikely, but possible errors in the mapping from AST nodes to the corresponding instructions in the intermediate representation for which the must-alias analysis is computed.

ing to the inter-procedural points-to analysis). This is the case in the following Java excerpt. The must-alias analysis is restricted to a single method. It will therefore never report that the arguments `a` and `b` of the invocations in method `m(Object,Object)` must alias. The inter-procedural points-to analysis, in contrast, will report that the arguments may alias:

```
1 void m(Object a, Object b) {          void caller() {
2   System.out.println(a);                Integer o = new Integer(8);
3   System.out.println(b);                m(o,o);
4 }                                     }
```

Both the points-to analysis and the must-alias analysis originate from the SOOT Java Optimization Framework [VRCG+99]. They are computed for its JIMPLE intermediate representation of the program (cf. Figure 2.4). To determine whether two expressions are in a may-alias or must-alias relation, the unification procedure therefore has to map each AST node to the corresponding instruction in the intermediate representation. Section 6.7 discusses this mapping in the context of the open-ended implementation of the procedure.

As the least restrictive domain-specific extension to the unification procedure, it is only attempted when the other extensions have failed.

Unifying names of the non-variable kind are instances of `Name` that denote entities of the non-variable kind (e.g. a method and a class) according to the semantic analysis and have identifier strings that unify (i.e. equal strings). For each name, the semantic analysis determines which entity it denotes.

The semantic analysis discerns four important kinds of entities: packages, types, methods and variables. Entities of the variable kind include fields, local variables, parameters and exception variables in catch clauses. A `Name` that denotes an entity of the variable kind is either used as an expression or is the "name" part of the declaration that declares the entity.[10] The former names are treated by the unification procedure as expressions, while the latter names are treated as variable declaration names. A query that requires such names to unify expresses a data flow characteristic. Unifying such names because their identifier strings are equal would result in false positives of the data flow characteristic (cf. Section 5.3.4). This domain-specific unification extension is therefore only applicable to `Name` instances that denote an entity of the non-variable kind.

A query that requires `Name` instances of the non-variable kind to unify, expresses a syntactic characteristic. No false positives can result. The following query, for instance, identifies non-constructor methods that are named after the class they are declared in:

```
1 if ?c definesMethod: ?m,
2    ?c classDeclarationHasName: ?name,
3    ?m methodDeclarationHasName: ?name,
4    [?m isConstructor not]
```

[10] The AST for method `void m(Integer i){Object j = i;}` has 4 `Name` instances representing `m`, `i` (in the parameter list), `i` (in the body) and `j` respectively. The `Name` instance for `m` denotes an entity of the non-variable kind (i.e. a method). The name instances for `i` in the parameter list and for `i` in the body both denote an entity of the variable kind (i.e. a formal parameter). The instance in the parameter list is a variable declaration name, while the instance in the body is a name that is used as an expression.

The `Name` instances in the above query denote entities of different non-variable kinds. `Name` instances that denote different entities of the same non-variable kind (e.g. two methods) can also be required to unify. For instance, in example-based specifications for overriding and overloading methods (cf. Chapter 8). Section 6.6.2 demonstrates such a unification in an LMP specification for overriding methods.

A variable declaration name and an expression that unify are an instance of `Name` and `Expression` respectively such that the former is a variable declaration name (i.e. the "name" part of a `VariableDeclaration` instance) and the latter is in a may-alias (unification degree of $\frac{1}{2}$) or must-alias (unification degree of $\frac{9}{10}$) relation with the entity declared by the variable declaration. This entity can be a field, local variable, formal parameter or exception variable in a catch clause.

The above would suffice for a pure object-oriented language. For Java expressions of a primitive type, however, aliasing information is not available. We therefore refine the above with the results of the semantic analysis.[11] This analysis relates names to the entities they denote. The names of a field and parameter declaration unify (unification degree of $\frac{9}{10}$) with a variable reference that respectively references the declared field and parameter according to the semantic analysis. In method `m(int p)`, for instance, the right-hand side of expression `x = p` is a reference to parameter `p`. Variable references (i.e. `FieldAccess` instances and `Name` instances of the variable kind) are the only expressions for which semantic analysis information is available.

The refinement of this domain-specific unification extension is *not* applicable to local variable and exception variable references. These can be assigned in the body of a method. The semantic analysis is oblivious to such assignments (i.e. it only relates a variable reference to the corresponding declaration).

The second query in Figure 4.10 expressed the data flow characteristics of the getter method through the occurrences of variable *?name* on lines 9 and 10:

```
6    ...
7    ?fieldDeclaration fieldDeclarationHasFragments: ?fragments,
8    ...
9    ?fragments contains: variableDeclarationFragment(?name, ?, ?),
10   ?method methodDeclarationHasBody: block(nodeList(<returnStatement(?name)>))
```

The query requires the operand of the return statement in a getter method to unify with the field it protects. The `Name` instance bound to *?name* on line 9 is always the "name" part of a variable declaration because `VariableDeclarationFragment` extends `VariableDeclaration`.[12] On line 10, variable *?name* is bound to an expression. The bindings for *?name* on lines 9 and 10 are therefore unified by this domain-specific unification extension.

[11] To ensure that two AST nodes always unify with the maximum unification degree, the semantic analysis is consulted before the alias analyses.

[12] The subclasses of `VariableDeclaration` are `SingleVariableDeclaration` and `VariableDeclarationFragment`. The former represent formal parameters and exception variables in catch clauses. The latter represent the fragments of field declarations which can declare multiple fields. The same goes for local variable declarations (e.g. fragment `j=1` in declaration `int i=0, j=1;`).

A variable declaration and an expression unify according to a variation of this domain-specific unification extension. The following conditions can therefore substitute for the conditions on lines 9–10 of the getter method query in Figure 4.10:

```
8     ...
9     ?fragments contains: ?f,
10    ?method methodDeclarationHasBody: block(nodeList(<returnStatement(?f)>))
```

Unifying variable declaration names are two `Name` instances that are each the "name" part of a variable declaration that declares an entity of the variable kind (i.e. a field, local variable, parameter or exception variable). Both entities have to be of a reference type and in a may-alias (unification degree of $\frac{1}{2}$) or must-alias (unification degree of $\frac{9}{10}$) relation with each other.

Note that the variable declaration parents of two unifying variable declaration names need not unify and vice versa. There is only one domain-specific unification extension that concerns two variable declarations: formal parameters unify if their types unify. The names of the parameters need not unify. Other variable declarations (e.g. a field declaration) do not unify unless they are the same AST node.

A method invocation name and method declaration name that unify are the "message" part of a method invocation and the "name" part of the corresponding method declaration respectively. Both are instances of `Name`.

The associated unification degree is $\frac{1}{2}$ if the invocation may invoke the method according to an approximation of the dynamic type of its receiver (i.e. the union of the dynamic types of all heap object approximations in its points-to set). This is also how CAVA's predicates for control flow information resolve polymorphic method invocations (cf. Section 5.2.1).

If the method invocation and declaration do not unify according to the above, the associated unification degree is $\frac{1}{4}$ —provided the invocation may invoke the method according to the static type of its receiver (i.e. class hierarchy analysis [DGC95]). Otherwise, the unification fails.

While the latter unifications are less precise with respect to the run-time behavior of the program, they are often required in example-based specifications. For instance, when the target method declaration is exemplified as an interface method or as an abstract class method. Such methods are not invoked at run-time because the dynamic type of the invocation's receiver implements them. We will illustrate this in Section 8.1 of the validation chapter.

A method invocation (name) and a method declaration (name) unify according to variations of this domain-specific unification extension.

Unifying types are instances of `Type` (fully qualified, unqualified, parametrized, etc.) that denote the same type according to the semantic analysis.

The introduction to this section illustrated how an unqualified type `List` can unify with the qualified type `java.util.List` if the unit in which it resides has an import declaration `java.util.List;` or `java.util.*`.

Unifying return types denote the same type *or* are in a sub-type relation where the method declaration of the sub-type overrides the method declaration of the

super type (all checked using the semantic analysis). This keeps the unification procedure consistent with the Java semantics which supports co-variant return types. Return type AST nodes are instances of `Type` that are the "return type" part of a method declaration. Section 6.6.1 will illustrate this domain-specific unification.

Note that the overriding condition ensures that the relation of unifying AST nodes is *symmetric* (i.e. $xRy \Rightarrow yRx$) even though the sub-type relation is not. Otherwise, condition *?x* `equals:` *?y* would allow *?x* to be a sub-type of *?y*, while condition *?y* `equals:` *?x* would allow *?y* to be a sub-type of *?x*.

A type and type declaration that unify are an instance of `Type` (e.g. `java.util.List`) and `TypeDeclaration` (e.g. an interface declaration) such that the former AST node denotes the type declared by the latter AST node according to the semantic analysis.

The following logic rule identifies ad-hoc copy constructors. These are often the cause of subtle bugs involving a shallow copy where a deep copy was intended instead:

```
1 ?m isPossibleCopyConstructorIn: ?c if
2   ?c definesMethod: ?m,
3   ?c classDeclarationHasName: ?name,
4   ?m methodDeclarationHasName: ?name,
5   ?m methodDeclarationHasParameters: nodeList(<?p>),
6   ?p singleVariableDeclarationHasType: ?c
```

The rule requires the name of the method to unify with the name of its declaring class. This is checked by the domain-specific unification extension concerning names of the non-variable kind. In addition, the method should have a single parameter of the type declared by its class. This is checked by this domain-specific unification extension.

Reflections

The above domain-specific unification extensions ensure that implicit points of variation among pattern instances are recognized. Syntactically differing nodes can, for instance, denote the same type or evaluate to the same object at run-time. The extensions enumerated above are those that we implemented to validate our approach. More extensions may be needed in the future.

The domain-specific unification procedure cannot be bypassed in a query. If necessary, solutions that needed a domain-specific extension can be excluded by adding a condition to the query. Adding condition "*?x* `equals: ?y : [1]`" excludes solutions in which the unification degree of *?x* and *?y* is lower than 1 (i.e. possible false positives). Adding condition "`[`*?x* == *?y*`]`" excludes solutions in which *?x* and *?y* are not bound to the same AST node (i.e. solutions that would not be identified under the general-purpose unification procedure).

6.4.2 Unifying a Reified AST Node and a Logic Term

The previous section enumerated which combinations of two reified AST nodes unify. Here, we describe the combinations of a reified AST node and a logic term (other than a reified AST node) that unify:

An AST node and an uninstantiated compound term that unify are structurally equivalent instances of `ASTNode` and `CompoundTerm` respectively of which the latter has not yet been unified with any AST node before. A compound term $f(t_1,\ldots,t_n)$ is structurally equivalent to an AST node with children $\delta_1,\ldots,\delta_n$ if the term's functor f unifies with the decapitalized name of the node's class its multiplicity n has to agree with the amount of child nodes and each of the term's arguments t_i unifies with child node δ_i. The associated unification degree is the product of the unification degrees to which each t_i unifies with δ_i. This is in line with the unification degree for two composite terms (cf. Section 6.1.2).

The following query illustrates this domain-specific unification extension on an instance of `CastExpression`:

```
1  if ?cast isExpression,
2     [?cast isKindOf: JavaWorld.org.eclipse.jdt.core.dom.CastExpression],
3     ?cast castExpressionHasType: ?type,
4     ?cast castExpressionHasExpression: ?expression,
5     ?cast equals: castExpression(?type,?expression)
```

Unifying an AST node with a structurally equivalent compound term (line 5) is short-hand for a condition that restricts the type of the node (line 2) and subsequent unification conditions over its child nodes (lines 3–4). There is however more to an AST node than its type and child nodes. The structurally equivalent compound term has no information about the node's identity, the node's parent or the node's state (i.e. instance variables).

The unification procedure therefore distinguishes compound terms that have not yet been unified with an AST node from those that have. We refer to the former and the latter as *uninstantiated compound terms* and as *instantiated compound terms* respectively. Unifying an uninstantiated compound term with an AST node instantiates the term to the AST node. From then on, the instantiated compound term represents the AST node it is instantiated to. Each instantiated compound term is an alternative reified version of an AST node. Unifying two instantiated compound terms, for instance, amounts to unifying the AST node they are instantiated to. The following queries illustrate the need to instantiate compound terms to the AST node they are unified with:

```
1  if ?x isCastExpression,
2     ?y equals: castExpression(?t, ?e),
3     ?y equals: ?x,
4     [?y getType] equals: ?t
```

```
5  if ?x isCastExpression,
6     ?y equals: ?x,
7     ?y equals: castExpression(?t, ?e),
8     [?y getType] equals: ?t
```

Otherwise, the Smalltalk terms on lines 4 and 8 would evaluate to different objects —even though the queries only differ in the order of their conditions. In fact, the query on the left would raise an exception because the `CompoundTerm` instance bound to *?y* does not understand message `getType`. Instantiating compound terms to the objects they unify with safeguards the declarative nature of logic queries.

This domain-specific unification extension accommodates using compound terms to quantify over AST nodes. Without, users would have to resort to the second rather than the first query depicted in Figure 4.9. The second query illustrated the unification-related shortcomings of CAVA's identity-based reifi-

cation to objects under the general-purpose unification procedure (cf. Section 4.4.2).

The implementation of the extension invokes the API for structural reflection offered by the entire `org.eclipse.jdt.core.dom.ASTNode` hierarchy (cf. Section 5.2.1). The implementation is therefore generic (i.e. it is only implemented on the root class) and evolves with the parser and the language specification.

Unifying uninstantiated compound terms are instances of `CompoundTerm` of which the functors and corresponding arguments unify (i.e. regular Prolog unification).

An AST node and an instantiated compound term that unify are instances of `ASTNode` and `CompoundTerm` respectively such that the former unifies with the AST node the latter is instantiated to. The associated unification degree is the degree to which both nodes unify.

Because the term represents the node it is instantiated to, the term and the node are *not* required to be structurally equivalent. As a result, the following query has solutions in which the bindings for variables *?f1* and *?f2* differ:

```
1  if ?compound equals: ?f1@(?args1),
2     ?e1 isExpression,
3     ?compound equals: ?e1,
4     ?e2 isExpression,
5     (?compound equals: ?e2) : ?degree,
6     ?e2 equals: ?f2@(?args2),
7     not(?f1@(?args1) equals: ?f2@(?args2)),
8     (?e1 equals: ?e2) : ?degree
```

The first condition binds *?compound* to an uninstantiated compound term.[13] The third condition instantiates the term to an expression *?e1*. The fourth condition illustrates this domain-specific unification extension: the instantiated compound term *?compound* unifies with expression *?e2* to the extent *?degree*. This is the extent to which *?e1* and *?e2* unify on line 8. Lines 6–7 illustrate that node *?e2* does not have to be structurally equivalent to *?compound*.

The first and second argument to `equals:/2` on line 7 are uninstantiated compounds with the same functor and arguments as the instantiated compounds on line 3 and line 6 respectively. The uninstantiated compounds do not unify if their functors or one pair of corresponding arguments does not unify. The former is possible if *?e1* and *?e2* are bound to instances of different classes that unify according to one of the domain-specific extensions in the previous section. For instance, a `ParenthesizedExpression` and `CastExpression` that are in a may-alias relation. They are structurally equivalent to `parenthesizedExpression(?)` and `castExpression(?,?)` respectively.

Unifying instantiated compound terms are instances of `CompoundTerm` that are instantiated to AST nodes that unify. The associated unification degree is the degree to which both nodes unify. The functors and corresponding arguments of the term are *not* required to unify. The following query therefore has the same solutions as the previous query:

[13] Term `?f1@(?args1)` is a variable argument compound term (cf. Section 5.1.1).

```
1  if ?e1 equals: ?f1@(?args1),?e2 equals: ?f2@(?args2),
2     ?e1 isExpression, ?e2 isExpression,
3     ?e1 equals: ?e2,
4     not(?f1@(?args1) equals: ?f2@(?args2))
```

Line 3 illustrates this domain-specific unification extension: *?e1* and *?e2* are bound to compound terms that have been instantiated by on line 2 (reification predicate `isExpression/1` quantifies over all expressions).

An instantiated and uninstantiated compound term that unify are instances of `CompoundTerm` such that their functors and arguments unify. The unification procedure instantiates the uninstantiated term to the object represented by the instantiated term.

The following query illustrates this unification extension:

```
1  if ?e equals: castExpression(?,?), ?e isExpression,
2     ?m isMethodDeclaration, ?e isChildOf: ?m : ?degree

3  ?term isChildOf: ?functor@(?args) if ...
```

The conditions on the first line instantiate a compound term to a cast expression. The conditions on the second line identify a method declaration that has an expression that unifies with this cast expression to extent *?degree*. The unification extension is needed to unify the head of the rule that implements predicate `isChildOf:/2` (cf. Section 5.2.2) with the last goal in the query.

Note that method declaration *?m* is not necessarily the parent method declaration of the cast expression to which the compound *?e* is instantiated (i.e. `not(?m equals: [?e parentMethodDeclaration])` Against the program depicted in Figure 5.9, its solutions include bindings *?m*→"`Object e(){return a();}`", *?degree*→ $\frac{1}{2}$ with *?e* bound to a compound instantiated to the cast expression in method "`Date c(Object a){return (Date) a;}`". Section 6.6.4 clarifies why expression "`a()`" and expression "`(Date) a`" are in a may-alias relation.

An AST node and a regular expression term that unify are an instance of `ASTNode` and `RegExpTerm` respectively such that the regular expression represented by the latter matches the source code represented by the former.

The following query identifies method declarations that have the string `"Visitor"` in their body:

```
1  if ?m methodDeclarationHasBody: {.+Visitor.+}
```

The second argument to `methodDeclarationHasBody:/2` is a regular expression term. The regular expression itself is demarcated by braces. Section 6.6.2 demonstrates this domain-specific unification extension.

A collection of AST nodes and a compound term unify according to variants of the above above domain-specific unification extensions. For instance, an instance of `ASTNode$NodeList` unifies with an uninstantiated compound term `nodeList(?list)` of which the single argument unifies with a logic list of the same size containing the elements of the collection. The following query therefore identifies method invocations with two arguments that unify with a single expression:

```
1  if ?e isExpression, ?i methodInvocationHasArguments: nodeList(<?e,?e>)

2  ?i methodInvocationHasArguments: ?a if
3     ?i isMethodInvocation, ?i equals: methodInvocation(?,?,?,?a)
```

Reflections

Having enumerated the implemented domains-specific unification extensions, we conclude this section by discussing their impact on the reification predicates of the CAVA library (cf. Section 5.2.1). Predicates that quantify over the children of an AST node merely have to unify their argument with a structurally equivalent compound term. This is illustrated by the implementation of predicate `methodInvocationHasArguments:/2` depicted above. The impact on predicates that quantify over all AST nodes of a certain kind is more subtle. *Were* predicate `isMethodInvocation/1` implemented as follows, it would behave different under the domain-specific and general-purpose unification procedure:

```
1  ?i isMethodInvocation if [Soul.MLI allMethodInvocations] contains: ?i
```

The Smalltalk term evaluates to a collection of all `MethodInvocation` instances in the program's AST. The predicate would therefore succeed on any AST node that unifies with one of these instances. Under the domain-specific procedure, the AST node is not necessarily a `MethodInvocation` itself. This is problematic as the conditions "`?i isMethodInvocation, [?i isConstructor]`" could raise an exception. For instance, when `?i` is bound to a `CastExpression` that is in a may-alias relation with one of the method invocations. The first condition would moreover introduce choice points in the proof procedure when `?i` is already bound to a method invocation. This is undesirable out of performance considerations.

To avoid these problems *and* keep the behavior of the CAVA library consistent under both unification procedures, the actual implementation of the reification predicate only quantifies over method invocation instances. It succeeds without introducing choice points if it is given such an instance or a compound term instantiated to such an instance. It fails on other AST nodes and other instantiated compound terms. On an uninstantiated compound term[14], the predicate succeeds if there is a method invocation that unifies with the uninstantiated term. As a result, the predicate instantiates the term to each structurally equivalent expression upon backtracking.

6.5 Logic Meta Programming with Domain-Specific Unification

The two previous sections respectively defined how reified AST nodes unify and how a reified AST node unifies with a logic term. Here, we illustrate the complete domain-specific unification procedure on two logic meta programming queries.

Example 1: Detecting Overriding Methods

The query depicted in Figure 6.6 expresses the overriding relation between two methods in terms of their syntactic characteristics and the inheritance relation between their classes. Whether or not its conditions implement the overriding relation correctly, depends on the unification procedure under which they are resolved:

[14] The reification predicates discern instantiated compounds from uninstantiated compounds in a way similar to predicate `isInstantiatedTo:/2` (cf. Section 4.4.3).

```
1  if ?m isMethodDeclaration,
2     ?overrider isMethodDeclaration,
3     [?overrider getParent] extends: [?m getParent],
4     ?m equals: methodDeclaration(?,?,?,?type,?name,?params,?,?,?),
5     ?overrider equals: methodDeclaration(?,?,?,?type,?name,?params,?,?,?)
```

Figure 6.6: Quantifying over overriding methods through dom.-spec. unification.

- *Under the general-purpose unification procedure*, the last two conditions would either fail or implement the relation incorrectly. The former is the case for the identity-based reification of CAVA that maps each AST node to a unique term. The occurrences of variable *?name* on lines 4 and lines 5, for instance, can never be bound to the same `SimpleName` instance.

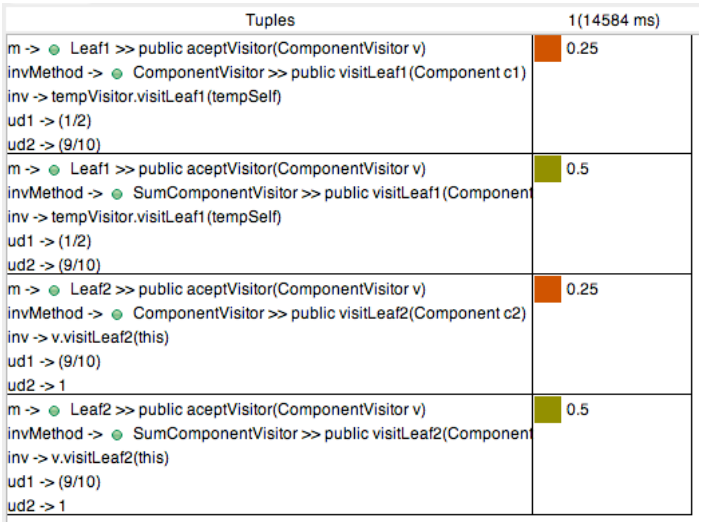
  The latter is the case for a reification that maps structurally equivalent AST nodes to the same term. Consider the parameters *?params* of the methods. The general-purpose unification procedure requires the names of corresponding parameters to unify. This is incorrect as Java only requires their types to be the same. The procedure moreover requires the AST nodes for the types to unify. This is incorrect as an unqualified type in one method does not necessarily denote the same unqualified type in the other method. Instances where one method uses an unqualified type (eg. `Object`) and the other a qualified type (e.g. `java.lang.object`) are also missed. The same goes for the return types of the methods. Worse, instances in which the return types are co-variant (i.e. the one of the overriding method is a subtype of the base method) are missed as well.

- *Under the domain-specific unification procedure*, the conditions implement the overriding relation correctly. Different `ASTNode$NodeList` instances (i.e. the occurrences of *?param*) unify if their corresponding elements unify. Instances of `SingleVariableDeclaration` (i.e. individual parameters) unify if the AST nodes for their types unify. Instances of `Type`, in turn, unify if they denote the same type according to the semantic analysis *or* if they are both the return types of methods that are in an overriding relation. Co-variant return types are therefore supported. The implementation of the unification procedure verifies this relation in the same manner as the implementation of predicate `overrides:/2` in the CAVA library (cf. Figure 5.12).

The domain-specific unification procedure enables expressing the structural "overriding" characteristic in terms of syntactic characteristics without introducing false positives or missing instances. Section 7.4.2 will show how this enables exemplifying such characteristics in example-based specifications. To understand the above query, however, one has to be familiar with the domain-specific unification procedure. In LMP specifications, it is therefore preferable to use a predicate that reifies the overriding relation explicitly (i.e. `overrides:/2`).

Example 2: Detecting Double Dispatching

Figure 6.7 depicts a query that identifies two methods *?m* and *?invMethod* such that the former invokes the latter through the idiomatic implementation of double



Tuples	1(14584 ms)
m -> Leaf1 >> public aceptVisitor(ComponentVisitor v) invMethod -> ComponentVisitor >> public visitLeaf1(Component c1) inv -> tempVisitor.visitLeaf1(tempSelf) ud1 -> (1/2) ud2 -> (9/10)	0.25
m -> Leaf1 >> public aceptVisitor(ComponentVisitor v) invMethod -> SumComponentVisitor >> public visitLeaf1(Component inv -> tempVisitor.visitLeaf1(tempSelf) ud1 -> (1/2) ud2 -> (9/10)	0.5
m -> Leaf2 >> public aceptVisitor(ComponentVisitor v) invMethod -> ComponentVisitor >> public visitLeaf2(Component c2) inv -> v.visitLeaf2(this) ud1 -> (9/10) ud2 -> 1	0.25
m -> Leaf2 >> public aceptVisitor(ComponentVisitor v) invMethod -> SumComponentVisitor >> public visitLeaf2(Component inv -> v.visitLeaf2(this) ud1 -> (9/10) ud2 -> 1	0.5

```
1  if ?m methodDeclarationHasParameters: nodeList(<?parameter>),
2     ?parameter singleVariableDeclarationHasType: ?type,
3     ?parameter singleVariableDeclarationHasName: ?param,
4     ?inv isChildOf: ?m, ?this isChildOf: ?m,
5     ?inv methodInvocationHasExpression: ?receiver,
6     (?receiver equals: ?param) : ?ud1,
7     ?inv methodInvocationHasArguments: nodeList(<?arg>),
8     ?this equals: thisExpression([nil]),
9     (?arg equals: ?this) : ?ud2,
10    ?invMethod isMethodDeclaration,
11    ?invMethod equals: ?inv
```

Figure 6.7: Quantified double dispatching implementations of Figure 5.4.

dispatching in single dispatch languages.[15] This implementation has method *?m* invoke method *?invMethod* on its single parameter passing its receiver (i.e. `this`) as an argument.

The query requires method *?m* to have a single parameter named *?param* of type *?type* declared in variable declaration *?parameter* (lines 1–3). The method has to have an invocation *?inv* (line 4) that invokes method *?invMethod* (line 11) on an expression *?receiver* that unifies with the parameter *?param* (line 6). The single argument *?arg* of this invocation should unify (line 9) with an unqualified `this`-expression *?this* (line 8) of method *?m* (line 4). The query relies on the domain-specific unification procedure: AST nodes unify with structurally equivalent compound terms (lines 1, 7 and 8), expressions unify with aliasing expressions (lines 6 and 9) and a method invocation unifies with the method declaration it may invoke (line 11).

The solutions to the query are depicted at the top of Figure 6.7. They stem from the program depicted in Figure 5.4 in which `Leaf1` and `Leaf2` are subclasses of `Component`. They implement the Composite design pattern [GHJV94].

[15] The query can be shortened by merging some explicit unification conditions. Condition `?inv equals: methodInvocation(?parameter,?,?,nodeList(<?this>))`, for instance, can substitute for the conditions on lines 5–7 and line 9. The conditions on lines 2–3 are not necessary either.

Class `ComponentVisitor` is the abstract root of a hierarchy of visitors for the `Component` hierarchy. It defines methods `visitLeaf1(Component)` and `visitLeaf2(Component)`. Class `SumComponentVisitor` extends `ComponentVisitor` and overrides these methods.

The first two solutions correspond to a method invocation *?inv* → `tempVisitor.visitLeaf1(tempSelf)` in `Leaf1`. Their truth degrees stem from the domain-specific unification of this method invocation with a method declaration *?invMethod* on line 11 (i.e. the minimum of the truth degrees for all goals in the query).[16]

The first solution has an associated truth degree of 0.25. This is because *?inv* → `tempVisitor.visitLeaf1(tempSelf)` only resolves to method *?invMethod* → `ComponentVisitor»visitLeaf1(Component)` according to the static type of tempVisitor. The second solution, in contrast, has an associated truth degree of 0.5. This is because the points-to analysis was able to determine the dynamic type `SumComponentVisitor` (i.e. a subclass of `ComponentVisitor`) for the receiver of *?inv*. Note that line 10 quantifies over all method declarations.

The remaining solutions correspond to a method invocation *?inv* → `tempVisitor.visitLeaf1(tempSelf)` in `Leaf2`. Again, the dynamic and static type of its receiver are `SumComponentVisitor` (truth degree 0.5) and `ComponentVisitor` (truth degree 0.25) respectively.

The unification degree *?ud2* (line 9) of the invocation's argument *?arg* and the `this`-expression *?this* differs for each binding of *?inv*. The first two solutions have binding *?ud2* → $\frac{9}{10}$ because *?this* and *?arg* are bound to local expressions that alias according to the intra-procedural must-alias analysis. The remaining solutions have binding *?ud2* → 1 because *?this* and *?arg* are bound to the same AST node and therefore unify according to the general-purpose unification procedure.

The unification degree *?ud1* (line 6) of the invocation's receiver *?receiver* and *?m*'s parameter *?param* differs for each binding of *?inv* as well. The first two solutions have binding *?ud1* → $\frac{1}{2}$ because `Leaf1` invokes `visitLeaf1` on a local that may-alias the parameter of method `aceptVisitor` (cf. Figure 5.4). The remaining solutions have binding *?ud1* → $\frac{9}{10}$ because `Leaf2` invokes `visitLeaf2` immediately on the parameter of method `aceptVisitor`.

The query requires nodes of the same kind (e.g. the arguments to *?arg* `equals:` *?this* are expressions) and nodes of different kinds to unify (e.g. the arguments to *?inv* `equals:` *?invMethod* are a method invocation and declaration). To understand the query, one has to know in which relation the nodes should be for their unification to succeed. For nodes of the same kind, this relation is straightforward because it is based on the semantics of the programming language (e.g. a qualified and unqualified type should denote the same type). The required relation between nodes of different kinds is less straightforward. It is therefore preferable to use a reification predicate for the required relation instead (e.g. *?inv* `mayInvoke:` *?invMethod*). In example-based specifications, however, multiple occurrences of the same variable express such relations naturally. [17].

[16] Recall that truth extracting goals succeed with a truth degree of 1 (e.g. lines 6 and 9).

[17] Figure 8.12 depicts an example-based specification for the double dispatching idiom.

6.6 Revisiting LMP Support for Pattern Characteristics

Having discussed the instantiations of the fuzzy logic and domain-specific unification cornerstones, we demonstrate how they improve upon the support offered by the LMP cornerstone for each kind of pattern characteristic (cf. Section 5.3).

6.6.1 Expressing Syntactic Characteristics

Figure 5.6 depicts a regular SOUL query that specifies the syntactic characteristics of a potentially enhanceable `for`-statements. The query depicted in Figure 6.8 has the same solutions (bottom of Figure 5.6), but is substantially shorter (i.e. 5 conditions versus the original 16). It relies on the domain-specific unification procedure.

The second condition in the improved query replaces the 3 "... has ..." conditions on lines 2–4 of the original query. This condition unifies the reified `for`-statement bound to *?for* with a structurally equivalent compound term to access its expression, updaters and body children. This was not possible under the general-purpose unification procedure because the CAVA library reifies the `for`-statement as the statement itself rather than a compound term (cf. Section 5.2).

The condition on line 5 is equivalent to conditions 12–16 of the original query. It restricts the bindings for *?nextInv* (a child of either the body or the updaters of the `for`-statement) to invocations of a method named `next` with an empty argument list. The condition achieves the former by unifying its name part (an instance of `SimpleName`) with compound term `simpleName(['next'])` and the latter by unifying its arguments part (an instance of `ASTNode$NodeList`) with `nodeList(<>)`. The original query restricted the size of the argument list by sending it message `size` in a Smalltalk term (line 16).

Evaluation As discussed in Section 5.3.1, the convoluted sequences of "... has ..." conditions of the original query evidence the unification-related shortcomings of LMP (cf. Section 4.4.2). They are no longer present in the improved query. This illustrates how the domain-specific unification procedure helps overcome the unification-related shortcomings of LMP.

Both queries use AST traversals to express the pattern's syntactic characteristics. As discussed in Section 5.3.1, the traversals evidence the quantification-related shortcomings of LMP. In Section 7.4.1, we will exemplify these characteristics through a source code excerpt instead. This will illustrate how the example-based specification cornerstone helps overcome the quantification-related shortcomings of LMP.

The improved query does not yet state that methods `hasNext()` and `next()` should be invoked on the same iterator. The false positive in method `not_enhanceable_1` is therefore still included in its solutions. In Section 6.6.4, we will express this data flow characteristic simply by requiring *?hnRec* and *?nRec* to unify according to the domain-specific unification procedure.

6.6.2 Expressing Structural Characteristics

This chapter does not improve on the LMP support for the structural characteristics of the AMBIENTTALK-specific coding convention described in Section 5.3.2. The query in Figure 5.8 already expresses the characteristics succinctly without reporting false positives. It relies on CAVA's reification predicates for structural in-

```
1  if ?for isStatement,
2     forStatement(?,?c,?updaters,?body) equals: ?for,
3     methodInvocation(?hnRec,?,simpleName(['hasNext']),nodeList(<>)) isChildOf: ?c,
4     or(?nextInv isChildOf: ?body,?nextInv isChildOf: ?updaters),
5     ?nextInv equals: methodInvocation(?nRec,?,simpleName(['next']),nodeList(<>)),
```

Figure 6.8: Syntactic char. of enhanceable `for`s with dom. -spec. unification.

formation (cf. Section 5.2.1) and basic reasoning predicates that connect syntactic and structural information (cf. Section 5.2.2). The relational nature of logic programming facilitates quantifying over these predicates (cf. Section 4.2.2).

The unification-related shortcomings of LMP are only evidenced by the conditions that express the syntactic characteristics of the coding convention. The domain-specific extension that unifies AST nodes with compound terms enables substituting condition `?m methodDeclarationHasName: simpleName(?id)` for the two conditions on lines 6–7. Alternatively, the following condition could substitute for the conditions on lines 6–8. It relies on the domain-specific extension that unifies an AST node with a regular expression that matches its concrete syntax:

```
1  ?m methodDeclarationHasName: {(meta|base)_.+}
```

6.6.3 Expressing Control Flow Characteristics

To express the syntactic characteristics of the protocol described in Section 5.3.3, the queries in Figure 5.10 already unify AST nodes with structurally equivalent compound terms. For instance, to identify invocations of method `c()` in the control flow of methods that comply with the protocol (line 10 of the query in the bottom-left corner of the figure). The sequences of "... has ..." conditions that are required without this extension would have distracted from the control flow characteristics of the protocol. Neither the domain-specific unification procedure, nor fuzzy LMP improves the support of regular LMP for the control flow characteristics of the protocol. The next section will demonstrate the former's support for the data flow characteristics of the protocol.

6.6.4 Expressing Data Flow Characteristics

We express the data flow characteristics of the pattern that describes potentially enhanceable `for`-statements and the pattern that describes methods complying with the protocol described in Section 5.3.3 separately.

Enhanceable `for`-Statements Revisited

Neither the regular LMP query in Figure 5.6, nor the equivalent query in Figure 6.8 which relies on domain-specific unification express the data flow characteristics of enhanceable `for`-statements. These require methods `hasNext()` and `next()` to be invoked on the same iterator.

Section 5.3.4 attempted to express this in terms of syntactic characteristics because the CAVA library does not provide reification predicates for data flow information. Concretely, the following conditions were added:

```
1     ?hasNextReceiver simpleNameHasIdentifier: ?id,
2     ?nextReceiver simpleNameHasIdentifier: ?id
```

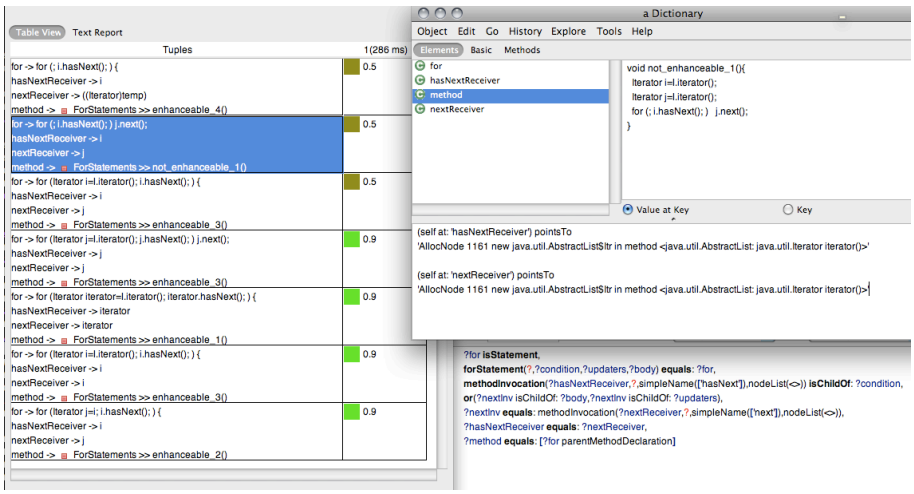


Figure 6.9: Data flow char. of enhanceable `for`s with dom. -spec. unification.

These conditions require the receivers of the `hasNext()` and `next()` invocations to be `SimpleName` nodes with unifying identifier strings. Figure 5.11 depicts the solutions to the query. Most notably, its solutions do not include the enhanceable `for`-statements in method `enhanceable_2` and method `enhanceable_4`. The receivers of the invocations in each method differ syntactically (`"i"` versus `"j"` and `"(Iterator) temp)"` versus `"i"`), but nevertheless evaluate to the same iterator.

The query in the bottom-right corner of Figure 6.9, in contrast, expresses this data flow characteristic by requiring both receivers to unify according to the domain-specific unification procedure (without restricting them to the `SimpleName` kind):

```
1    ?hasNextReceiver equals: ?nextReceiver
```

This condition extends the query in Figure 6.8.[18] The solutions to the extended query are shown in the top-left corner of Figure 6.9. We evaluated the query in fuzzy SOUL to quantify its solutions.

In solutions with a truth degree of $\frac{9}{10}$, *?hasNextReceiver* and *?nextReceiver* are bound to expressions that alias during every program execution according to the must-alias analysis. In solutions with a truth degree of 0.5, they are bound to expressions that alias during at least one program execution. Such solutions are more likely to be false positives. The query does not miss any of the previously unidentified enhanceable for-statements in methods `enhanceable_2` and `enhanceable_4`. The truth degree of the latter solution indicates that *?hasNextReceiver* and *?nextReceiver* are in a may-alias relation. However, the solutions to the query include two false positives. Both have an associated truth degree of 0.5.

The first false positive is the single occurrence of method `enhanceable_3` with a truth degree of 0.5. Its binding for *?nextReceiver* originates from the inner loop, while variable *?for* is bound to the outer loop. As in the original query, this is because the AST traversal on line 3 does not properly constrain the nesting of *?nextInv*

[18] In the original query, *?hasNextReceiver* and *?nextReceiver* have been shortened to *?hnRec* and *?nRec* respectively.

within *?body* (cf. Section 5.3.1). Because local variables `i` and `j` point to different iterators at run-time, one however expects that unifying *?hasNextReceiver* and *?nextReceiver* would have eliminated this solution.

Both false positives are reported because of imprecise points-to analysis results in our program representation. The highlighted occurrence of method `not_enhanceable_1`, the second false positive, will clarify this. The points-to sets for `i` and `j` are printed on the second and fourth line at the bottom of the inspector window in the top-right corner of Figure 6.9. As the intersection of their points-to sets is non-empty, they are in a may-alias relation according to the points-to analysis. The must-alias analysis was unable to assert that they are in a must-alias relation. The unification procedure therefore unifies *?hasNextReceiver* and *?nextReceiver*, but only with a unification degree of 0.5 to indicate that it might introduce false positives. This is the case for this solution because the points-to sets for `i` and `j` are imprecise. They contain a single heap referee abstraction that is allocated in method `iterator()` of `AbstractList`. They are computed by the SPARK [Lho02] analysis which is context-insensitive. It does not analyze individual invocations of a method in different invocation contexts separately (cf. Section 2.5.4). According to the analysis, all invocations of the `iterator` method return the same heap referee abstraction:

```
1 public Iterator iterator() { return new Itr(); }
```

The domain-specific unification procedure of our prototype will therefore always unify `AbstractList` iterators because its imprecise points-to analysis considers them in a may-alias relation.

Evaluation The reported pattern instances illustrate that whole-program analyses enable detecting implicit implementation variants of data flow characteristics. This obviates the need for specifications that enumerate these variants in an ad-hoc manner. Incorporating whole-program analyses in the unification procedure allows data flow characteristics to be specified without having to refer to the analyses that enable their detection. In particular, they can be expressed through multiple occurrences of a variable or by requiring different variables to unify. The latter is illustrated by the pattern specification.

To eliminate the reported false positives, a context-sensitive points-to analysis can be adopted —in particular one that models invocation contexts using the receivers of the topmost invocations on the call stack [LH06]. However, the fuzzy logic cornerstone already allows discerning solutions that required a unification which could introduce false positives. In this way, both cornerstones are complementary.

The Protocol Revisited

The protocol described in Section 5.3.3 requires complying methods to initiate the protocol by invoking `a()` and subsequently use the returned result as the argument for an invocation of `c(Object)` without invoking `b()` in between. The left side of Figure 5.10 depicts two queries that detect complying methods. The query in the top-left corner expresses their control flow characteristics in terms of syntactic characteristics, while the query in the bottom-left corner expresses them directly. Their results are shown in the top-left window in Figure 5.9.

The CAVA library does not provide reification predicates for data flow information. In an attempt to express the pattern's data flow characteristics in terms of

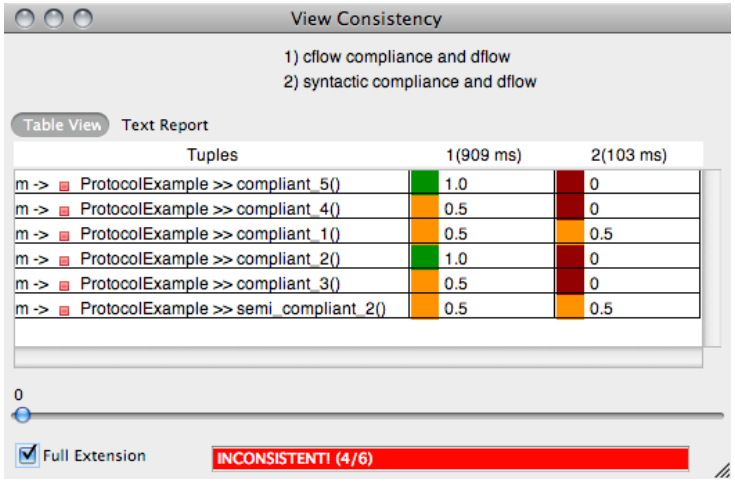


Figure 6.10: Quantified solutions for data flow char. of complying methods.

syntactic characteristics, Section 5.3.4 therefore extended the queries as follows (*?c* and *?a* are bound to invocations of methods `a()` and `c(Object)` respectively):

```
1  ?c methodInvocationHasArguments: ?args, [?args size = 1], ?args contains: ?a
```

This query requires invocation *?a* to be the actual argument of invocation *?c*. The query therefore failed to recognize methods `compliant_1`, `compliant_3` and `compliant_4` of Figure 5.9. Each method implements a different implicit variation point of the data flow characteristic. The latter two can only be recognized by analyzing the entire program.

The domain-specific unification procedure allows expressing data flow characteristics without exposing users to the whole-program analyses that enable detecting their implicit variation points. Only the condition on line 10 of either the top-left query (expressing control flow characteristics indirectly) or the bottom-left query (expressing control flow characteristics directly) in Figure 5.10 has to be changed:

```
1  ?c equals: methodInvocation(?,?,simpleName(['c']),nodeList(<?a>))
```

The condition succinctly expresses that the single argument of method invocation *?c* should be the result of method invocation *?a*. The quantified results for both adapted queries are depicted in Figure 6.10. The left and right column respectively list the truth degrees associated with solutions to the query that expresses the control flow characteristics directly and indirectly.

The first column in Figure 6.10 depicts the truth degrees for the solutions to the extended query that expresses the control flow characteristics directly. Apart from method `semi_compliant_1` (cf. Section 5.3.3), it identifies all complying methods and does not identify a false positive. For solutions with a truth degree of 1, the argument of the invocation of `c(Object)` is the actual invocation of method `a()` itself.

The second column illustrates the difficulties that arise in mapping AST nodes to the intermediate program representation commonly used by program analyses. It depicts the solutions to the extended query that fails to express the control

flow characteristics properly. Of the complying methods, the query only identifies methods `compliant_1` and `semi_compliant_2`.[19] Compared to the original query, it no longer reports the false positive `not_compliant_1` (cf. the second column of the top-left window in Figure 5.9). As the query still expresses the control flow characteristics in terms of syntactic characteristics, this improvement is due to the domain-specific unification procedure. Method invocation `a()` in the `true`-branch (bound to *?a*) does not unify with the argument of method invocation `c(d)`.

However, the reason is not immediately clear. Because of the assignment in the `true`-branch, the return value of `a()` should have been included in the points-to set for local variable `d`. A closer inspection reveals that the unification procedure could not query the points-to analysis results for the argument of `c(d)`. This invocation is absent from the JIMPLE intermediate byte code representation for which the analysis is computed (cf. Section 5.3.4). The Java compiler omitted the `if`-statement in the bytecode on lines 1–4 below. Lines 5–7 show the corresponding JIMPLE representation. Here, the code is further optimized such that the result of `a()` is not even stored:

```
1 aload_0 [this]
2 invokevirtual protocol.ProtocolExample.a() : example.Date [27]
3 astore_1 [d]
4 return

5 r0 := @this: protocol.ProtocolExample
6 virtualinvoke r0.<protocol.ProtocolExample: example.Date a()>()
7 return
```

For this false positive, the query benefits from the inherent incompleteness of the mapping from arbitrary AST nodes to the intermediate representations analyses are usually computed for (cf. Section 2.5.1). However, the false positive would have been included in its results if the compiler had not determined that the condition of the `if`-statement always evaluates to true. Method `c(Object)` is invoked after method `a()` according to the query (cf. Section 5.3.3).

Evaluation The above illustrates the difficulties that arise in mapping AST nodes to instructions in intermediate representations. Possible errors in this mapping warrant the unification degree of $\frac{9}{10}$ rather than 1 for expressions that are guaranteed to evaluate to the same object according to the must-alias analysis. Under the general-purpose unification procedure, users would have to implement this mapping and the domain-specific unification extensions that require them repeatedly in logic queries.

6.7 Open Implementation

The fuzzy variant of SOUL extends the object-oriented implementation of SOUL. It is therefore as open as SOUL itself (cf. Section 5.4). The same goes for its implementation of the domain-specific unification procedure. The interpreter determines whether two logic terms unify by sending the message `unifyWith:inEnv:myIndex:hisIndex:inSource:` to the Smalltalk implementation of one of the terms with the Smalltalk implementation of the other term as

[19] Methods with a truth degree of 0 are not included in its solutions, but are included in the entries of the right column to facilitate comparing its solutions with those of the other query. In order for a goal to succeed, its truth degree has to lie within]0,1] (cf. Section 6.1).

argument. The corresponding methods comprise the meta-level interface through which the unification procedure can be extended.

Our domain-specific extensions implement a double dispatching idiom on the `org.eclipse.jdt.core.dom.ASTNode` hierarchy. The methods at the top of Figure 6.11, for instance, implement the extension that determines whether two types unify. The first method is invoked by the interpreter on a `Type` instance. It double dispatches to the second method if its argument is another `Type` instance. Two type nodes unify if they are the same instance of `Type` (cf. Section 6.4.1). This is checked by the unification procedure on line 5. Two type nodes also unify if they denote the same type according to the semantic analysis. The unification procedure verifies this on lines 6–8 by comparing the "bindings" for the type nodes (i.e. entries in the symbol table of the Eclipse compiler —cf. Section 5.3.4). Return types of overriding methods also unify if the return type of the overriding method denotes the same or a sub-type of the return type of the overridden method. The implementation only checks whether their method declaration parents are in an overriding relation (lines 9–14). It does not check whether the return types are co-variant. This is unnecessary as the semantic analysis already asserted the overriding relation.

To facilitate implementing custom unification extensions, we provide a tool through which developers can edit a matrix that defines how each pair of AST nodes unifies (see [BDM07] for a screenshot). The extension itself has to be implemented in Smalltalk, but the tool automatically generates the remaining code for the double dispatching and ensures the symmetry of the algorithm. This way, the developer can concentrate on the implementation of the extension itself.

The SOOT Java Optimization Framework [VRCG+99] computes its points-to and must-alias analyses for the JIMPLE three-address based representation (cf. Figure 2.4). The instructions in this representation take the form of at most two operands, an operation and a result. We provide an API through which the JIMPLE instruction that corresponds to an expression in an ECLIPSE AST node can be retrieved. Custom unification extensions can therefore query other program analyses for their results for such an instruction.

The method at the bottom of Figure 6.11 implements the mapping from `CastExpression` instances to JIMPLE instructions. It returns an association of a unit (i.e. statement) and a local (i.e. variable) from the JIMPLE method that corresponds to the method declaration in which the expression resides (line 2).[20] The local is assigned the value of a JIMPLE cast expression by the unit (i.e. it is the left hand side of an assignment unit). The `JCastExpr` and the `CastExpression` have to cast to the same type. We consult the semantic analysis to determine to which type the latter casts (line 4). Their respective operands should also be compatible (line 5). The operand of the `CastExpression` is one of its child nodes. Its parent node should also be compatible with one of the trailing units that use the result of this local.[21] If the information about the parent and child nodes of the expression does not suffice to unambiguously identify its corresponding unit, the procedure also checks whether their line numbers correspond (line 7).[22] The mappings for other expressions are similar, but complicated. For instance, because of synthetic fields, methods and constructor arguments generated by the Java compiler to im-

[20] Cast expressions in field declarations are handled similarly.

[21] Consider the unit that assigns a local which corresponds to a cast expression that is used as the argument to a method invocation. It should have a trailing unit that uses the local as the argument to a corresponding method invocation.

[22] These are only available when the program has been compiled with debug information.

```
Type>>unifyWith: aTerm inEnv: anEnv myIndex: myIndex hisIndex: hisIndex inSource: aBool
  ^aTerm unifyWithType: self inEnv: anEnv myIndex: hisIndex hisIndex: myIndex inSource: aBool not.

Type>>unifyWithType: aTerm inEnv: anEnv myIndex: myIndex hisIndex: hisIndex inSource: aBool
  | selfbinding otherbinding |
  self = aTerm ifTrue: [^true].
  otherbinding := aTerm resolveBinding.
  selfbinding := self resolveBinding.
  (selfbinding isEqualTo_IBinding: otherbinding) ifTrue: [^true].
  (self isReturnTypeOfMethodDeclaration and: [aTerm isReturnTypeOfMethodDeclaration])
    ifTrue: [| myMethodBinding otherMethodBinding |
            myMethodBinding := self getParent resolveBinding.
            otherMethodBinding := aTerm getParent resolveBinding.
            ^(myMethodBinding overrides_IMethodBinding: otherMethodBinding)
              or: [otherMethodBinding overrides_IMethodBinding: myMethodBinding]].
  ^false
```

```
MLIForSoot>>localInUnitForCastExpression: aCastExpression
  ^(aCastExpression parentMethodDeclaration correspondingSootMethod)
    localInUnitIsAssignedValueSatisfying:
      [:value | (value isKindOf: JavaWorld.soot.jimple.internal.JCastExpr) and:
       [(self sootType: value getType isCompatibleWithBinding: aCastExpression resolveTypeBinding) and:
        [self subExpression: aCastExpression getExpression isCompatibleWithLocal: value getOp]]]
    byUnitSatisfying: [:unit | self node: aCastExpression isPositionCompatibleWithUnit: unit]
    andSomeTrailingUnitSatisfying:
      [:unit :local :unitChain |
       self sootUnit: unit usesLocal: local asParentOfExpression: aCastExpression inUnitChain: unitChain]
```

Figure 6.11: A domain-specific unification extension (top) and a method mapping Eclipse AST nodes to instructions in the JIMPLE intermediate representation (bottom).

plement inner classes.

6.8 Limitations of the Instantiation

This section enumerates the technical limitations of fuzzy SOUL and the domain-specific unification procedure. We discuss the open research questions related to their respective cornerstones in the concluding chapter.

Fuzzy Logic Programming using Fuzzy SOUL

In our prototype, the fixed quantification of logic connectives and resolution is arguably too restrictive for general-purpose fuzzy logic programming. Users can override this quantification only in a highly operational and ad-hoc manner using variable annotations and Smalltalk terms that manipulate truth degrees (cf. Section 6.1.1).

To better accommodate fuzzy logic programming, fuzzy SOUL could be generalized to quantify truth using custom t-norms [Háj98] (i.e. a generalized interpretation of conjunction from which interpretations for disjunction, negation and implication follow) and use truth degrees that generalize the unit interval (i.e. to unions of sub-intervals such as [VGMH02] or custom truth lattices). In future work, we want to investigate whether there are also configurations in this design space that improve the current ranking of pattern instances.

Our goal-driven resolution procedure does not aggregate identical solutions with different truth degrees into a single solution with an aggregated truth degree (cf. Section 6.1.2). Users have to aggregate over results manually. This further hinders the use of fuzzy SOUL to solve fuzzy problems. Straccia [Str06] provides an interesting starting point to address this deficiency in an efficient manner. It uses tabled resolution [RC97, CW96] which is already included in the future work for SOUL (cf. Section 5.5.1) and also addresses the aforementioned generalizations by allowing arbitrary functions in rules to manipulate values from truth lattices.

Premature Combination of Resolution Degree with Unification Degree

Section 6.1.2 describes how fuzzy SOUL computes the truth degree for a goal by multiplying the degree to which the goal and the head of a rule unify (i.e. unification degree) with the degree to which the goal can be resolved using this rule (i.e. resolution degree). The following program illustrates that unification and resolution degrees are sometimes combined prematurely:

```
1 (?y foo: ?y) : [0.7].

2 if ?e1 expressionMayAlias: ?e2,
3    (?x foo: ?e1) : ?t11,
4    (?x equals: ?e2) : ?t12

5 if ?e1 expressionMayAlias: ?e2,
6    (?x equals: ?e2) : ?t21,
7    (?x foo: ?e1) : ?t22
```

Predicate `foo:/2` is a weighted variant of `equals:/2`, while `expressionMayAlias:/2` reifies the may-alias relation between expressions. Although both queries only differ in the order of their last two conditions, the

truth degrees for solutions to both queries would differ significantly if the variable annotations on these conditions were removed (i.e. 0.5 for the first query and 0.35 for the second query). The bindings for these annotations are *?t11* → 0.7, *?t12* → 0.5 for the first query and *?t21* → 1, *?t22* → 0.35 for the second query.

At first sight, this problem could be addressed using different quantifications. Using minimum to combine unification and resolution degrees would ensure that the solutions to both queries have the same truth degrees (i.e. 0.5). However, all annotations of `foo:/2` that are above 0.5 would no longer have an influence on the truth degree of these solutions. We therefore use multiplication rather then minimum to ensure that both resolution and unification degrees influence the truth degree for a goal. Using multiplication to quantify conjunction would, on the other hand, result in very small truth degrees if the same condition is repeated in a query. This is because multiplication is not idempotent. The truth degree of the following query would for instance have a truth degree of 0.343:

```
1 if ?e1 foo: ?e2, ?e1 foo: ?e2, ?e1 foo: ?e2
```

In future work, we should delay combining unification degrees and resolution degrees until a truth degree is requested (i.e. for goal annotations). This would address the above problem. Solutions to a query that rely on two unifications based on points-to analysis lower would, moreover, be ranked lower than solutions that only require one. This without having to adopt multiplication to quantify conjunction (argued against above).

The similarity-based unification procedure of LIKELOG [AF99] is an interesting starting point. It computes and propagates unification degrees separately from the resolution degrees. Our quantification of unification degrees is closer to Sessa's weak unification procedure [Ses02].

The Relation of Unifying AST Nodes is not an Equivalence Relation

Each domain-specific unification extension (cf. Section 6.4.1) unifies two reified AST nodes if they are in a particular relation. For instance, two expressions unify if they are in a may-alias relation. Not all relations are symmetric. This is the case for the may-invoke relation and the sub-type relation. However, the overall relation R of unifying AST nodes is reflexive (i.e. xRx) and symmetric (i.e. $xRy \Rightarrow yRx$). Both are important properties. Without the latter, condition *?x* `equals:` *?y* and condition *?y* `equals:` *?x* would allow different bindings for *?x* and *?y*.

We ensure symmetry by taking the context of AST nodes into account. For instance, by checking whether two `Type` instances represent the return types of overriding method declarations. In this case, the return type of the overriding method is allowed to be a sub-type of the return type of the overridden method. Other `Type` instances only unify if they denote the same type. CAVA's reification (cf. Section 5.2) facilitates querying an AST node for its context. The unification procedure invokes the following method to discern `Type` instances:

```
1 Type>>isReturnTypeOfMethodDeclaration
2   ^self getParent isMethodDeclaration
```

The relation R of unifying AST nodes is not transitive. The following query can have a solution <*?e1* → e_1,*?e2* → e_2,*?e3* → e_3> such that $P(e_1) \cap P(e_2) \neq \varnothing \wedge P(e_2) \cap P(e_3) \neq \varnothing \wedge P(e_1) \cap P(e_3) = \varnothing$ where $pt(e)$ is the points-to set for expression e. Clearly, the may-alias relation (which determines when two expressions unify) is not transitive. The following query can have solutions:

```
1  if ?e1 isExpression, ?e2 isExpression, ?e3 isExpression,
2     ?e1 equals: ?e2, ?e2 equals: ?e3, not(?e1 equals: ?e3)
```

Under the general-purpose unification procedure, R is an equivalence relation. Under the domain-specific unification procedure, R is only a dependency relation: it is reflexive and symmetric, but not transitive. In future work, we want to investigate the theoretical implications of this difference. In practice, we have not yet encountered any problems caused by this difference. The unification extensions address specific shortcomings of LMP in pattern detection and their implementation precludes other uses in queries.

6.9 Conclusion

This chapter discussed the instantiations of the fuzzy logic and domain-specific unification cornerstones. Concretely, we defined a fuzzy version of SOUL with domain-specific extensions to the general-purpose unification procedure.

Fuzzy logic is a logic of quantified truth. It enables quantifying pattern instances with the extent to which they exhibit the characteristics expressed in a specification. We defined the semantics of fuzzy SOUL using the meta-circular evaluator methodology and discussed key predicates from its standard library. Apart from how it combines resolution degrees with unification degrees, its quantification of truth is representative for many early "fuzzy Prolog" systems. Rather uncommon are its support for rules that are weighted by a variable and for defining the characteristic function of a fuzzy set through linguistic symbiosis. We have shown how patterns with vague classification boundaries can be specified in this manner.

The domain-specific unification cornerstone overcomes the unification-related shortcomings of LMP in pattern detection. Terms that do not unify under the general-purpose procedure, can unify under the domain-specific procedure. The procedure computes a unification degree that reflects the likelihood that such a unification introduces false positives.

To support the natural use of unification to quantify over AST nodes, reified AST nodes unify with structurally equivalent compound terms —even if the reified version of an AST node is not a compound term. This is the case for the CAVA library which uses an identity-based reification to objects through the linguistic symbiosis of SOUL.

To recognize implicit points of variation among pattern instances, reified AST nodes unify if they represent different implementations of the same characteristic. This obviates the need for specifications that enumerate these variants in an ad-hoc manner. CAVA's reification facilitates incorporating whole-program analyses in the unification of individual AST nodes. Most notably, expressions unify with a unification degree of $\frac{1}{2}$ if they are in a may-alias relation according to an inter-procedural points-to analysis. Unifying such expressions can introduce false positives if the expressions do not evaluate to the same object during all possible program executions. Expressions that reside in the same method unify with a unification degree of $\frac{9}{10}$ if they are guaranteed to alias during all executions according to an intra-procedural must-alias analysis. If both expressions are the same AST node, they unify with a complete unification degree.

To illustrate how both cornerstones improve upon regular LMP, we specified the patterns that are representative for each kind of pattern characteristic as fuzzy logic queries that rely on domain-specific unification.

CHAPTER 7

INSTANTIATING THE EXAMPLE-BASED SPECIFICATION CORNERSTONE

The instantiation of the example-based specification cornerstone extends fuzzy SOUL *with template terms. These wrap code excerpts that correspond to the prototypical implementation of a pattern's essential characteristics. Having introduced their syntax, we revisit the meta-interpreter for fuzzy* SOUL *to clarify how template terms are resolved. Template terms are matched against the program under investigation. Matches should exhibit the characteristics exemplified by the code excerpt of the template term. However, a single code excerpt can exemplify different pattern characteristics. An AST node therefore always matches a template term under a particular example-based interpretation of the excerpt. Under the control flow interpretation, for instance, the control flow characteristics of the source code excerpt exemplify the intended matches. We discuss three standard example-based interpretations. We demonstrate how this cornerstone improves upon the support for the different pattern characteristics offered by regular LMP. Finally, we discuss the meta-level interface through which additional example-based interpretations can be defined.*

7.1 Extending SOUL with Template Terms

The example-based specification cornerstone overcomes the quantification-related shortcomings of logic meta programming (cf. Section 4.2.2). It suffices to exemplify a pattern's characteristics by means of a source code excerpt, rather than expressing them by explicitly quantifying over the reified program representation. The details of the program information *and* its reification are hidden (criterion **CSL5**). The resulting specifications are highly descriptive (criterion **CSL2**) and their syntax is familiar to application programmers.

However, explicit points of variation among pattern instances can only be expressed (criterion **CSL3**) by introducing "cutouts" (i.e. variables) in the source code excerpt of a template term. Logic meta programming, in contrast, features connec-

tives (e.g. `and/n`, `or/n` and `not/n`) to constrain such variation points. It moreover features an expressive means for abstraction and reuse of specifications (criterion **CSL4**): defining predicates.

As both cornerstones are complementary on criteria **CSL2**-**CSL5** and criteria **CSL3**-**CSL4** (cf. Table 4.1), we extended SOUL with template terms. They can be used anywhere a regular logic term is allowed. Depending on which one is more convenient, either a logic-based or an example-based specification can be used. Section 7.3 will moreover illustrate that integrating template terms in a logic language enables template composition.

7.1.1 The Syntax of Template Terms in a Nutshell

We introduced the syntax of template terms in Section 4.3. In brief, they consist of a compound term followed by a Java code excerpt demarcated by braces:

```
1  if jtStatement(?st){ ?x = (?type) ?e; }
```

The above query contains a template term for a Java statement. It consists of a functor `jtStatement`, a single argument *?st* and a code excerpt delimited by braces. *The functor of the template term identifies the grammar rule adhered to by the code excerpt.*[1] This grammar describes the concrete syntax of Java —extended with logic variables and a minimum of non-native syntax. The above excerpt exemplifies an expression statement (i.e. a statement that wraps an expression). The expression assigns, to a left hand side *?x*, the result of a cast to a type *?type* of an expression *?e*. Within a code excerpt, logic variables stand for productions that originate from a non-terminal in the Java grammar. They indicate explicit points of variation among pattern instances.

Template terms can be used anywhere a regular logic term is allowed (e.g. as conditions in logic rules and queries or embedded within other terms). Section 7.1.2 details how template terms are resolved. For now, it suffices to say that they are matched against the AST nodes of the program under investigation. Matching AST nodes exhibit the characteristics exemplified by the source code excerpt of the template term. Backtracking over the term successively unifies each matching node with the argument of the term. Variables within the excerpt get bound as well. The bindings for multiple occurrences of the same variable have to unify according to the domain-specific unification procedure (cf. Chapter 6).

Two-Argument Template Terms

Matches for a template term exhibit the characteristics exemplified by its source code excerpt. However, a single code excerpt can exemplify multiple characteristics (cf. Section 4.3.2). An AST node therefore always matches a template term under a particular example-based interpretation. Under the syntactic interpretation, for instance, the syntactic characteristics of the source code excerpt exemplify the intended matches. Under the control flow interpretation, in contrast, the control flow characteristics of the source code excerpt exemplify the intended matches. Section 7.2 details the predefined example-based interpretations. They realize the example-based semantics of the template terms.

[1] The prefix `jt` of the functor discerns template terms for Java statements from those for Smalltalk statements (which start with the `st` prefix).

The template terms presented so far have a single argument. This argument unifies with an AST node that matches the excerpt under a particular example-based interpretation. All example-based interpretations are considered successively upon backtracking over the term. In case a second argument is provided, only the example-based interpretation named after this argument is considered.

The first condition of the following query therefore restricts the example-based interpretations for the template term to those named `syntactic` or `controlflow` (i.e. the elements of the logic list[2]):

```
1  if <syntactic, controlflow> contains: ?interpretation,
2     jtMethodDeclaration(?match,?interpretation){
3       ?modList ?type ?name(?paramList){ ?s1; ?s2; }
4     }
```

The matches for the term can differ considerately under each interpretation. Under the control flow interpretation, instructions *?s1* and *?s2* need not reside in method *?match*. They can reside in different methods invoked from *?match*. Under the syntactic interpretation, in contrast, only perfect matches for the term are reported. The points of variation among the matches are restricted to those indicated explicitly by the logic variables in the excerpt.

The domain-specific unification procedure ensures that bindings for multiple occurrences of a variable are consistent across all terms in a specification. Different implementations of a data flow characteristic therefore match a template term regardless of the interpretation under which it is resolved.[3] Section 7.2.4 discusses the implicit variation points that each interpretation supports uniquely.

Variable Suffixes as Parser Directives

The above template term contains two directives for the parser of its code excerpt. Variables that end with the `List` suffix stand for a collection of concrete syntax elements. Variables *?paramList* and *?modList* therefore stand for the modifier list and formal parameter list of method *?match* rather than a single modifier and parameter. In solutions to the term, both variables will be bound to a collection of AST nodes. Note that logic variables are dynamically typed. This naming convention is therefore merely a parser directive.

Another parser directive is available to disambiguate source code excerpts. The combination of concrete syntax and dynamically typed variables is often ambiguous (i.e. can be recognized by multiple grammar rules that each construct a separate AST). Consider the class declaration template term in the following query:

```
1  if jtClassDeclaration(?classDeclaration) {
2        class ?className {
3          ?modifier ?type ?methodName(?parameterList) ?statementList
4        }
5   }
```

The source code excerpt of the term is ambiguous. It can exemplify a class with a method with a single modifier *?modifier* (1), a class with a constructor with modifiers *?modifier* and *?type* (2), a class with an additional member *?modifier* in its

[2] Recall that logic lists are demarcated by angle brackets in SOUL (cf. Section 5.1.1).

[3] Where this is not desirable, a condition of the form "*?x* `equals:` *?y* `: [1]`" excludes solutions in which the unification degree of *?x* and *?y* is lower than 1. A condition of the form "`[`*?x* == *?y*`]`" excludes solutions in which *?x* and *?y* are not bound to the same AST node (cf. Section 6.4).

body and a method without modifiers (3), a class with an additional *?modifier* in its body and a constructor with a single modifier *?type* (4) and a class with additional declarations *?modifier*, *?type* and a constructor without modifiers (5).

Template terms support this ambiguity by considering all possible ASTs for a source code excerpt upon backtracking. This way, users are not burdened with having to disambiguate the source code excerpts. The intended AST is always considered. If necessary, however, variables can be suffixed with the name of a grammar rule. Such variables only stand for the concrete syntax elements produced by this rule. Substituting variable *?type::jtType* for variable *?type*, for instance, disambiguates the above template term. The `::jtType` suffix will ensure that variable *?type::jtType* only stands for concrete syntax elements produced by the `jtType` grammar rule.

A Minimum of Non-Native Syntax

To ensure the descriptiveness of example-based specifications (criterion **CSL2**), we limit concrete syntax departures in template terms to the bare minimum. Without the following, however, many example-based specifications would be less concise:

Non-native operator := unifies its left hand side (a logic variable) with the AST node that matches the concrete syntax on its right hand side.[4] In the following query, the `:=` operator binds *?inner* to an inner class named *?innerName* of the class *?class* named *?name* that matches the template term:

```
1  if jtClassDeclaration(?class,?interpretation){
2       class ?name { ?inner := [class ?innerName ?innerMemberList] }
3     }
```

The following query is equivalent and does not use the `:=` operator, but is less concise (and slower):

```
1  if jtClassDeclaration(?class,?interpretation){
2       class ?name { ?inner }
3     },
4     jtClassDeclaration(?inner,?interpretation){
5       class ?innerName ?innerMemberList
6     }
```

The first template term establishes a binding for the left hand side of the original `:=` operator: a member declaration *?inner* of *?class*. The second template term matches the right hand side of the original `:=` operator and unifies this with its first argument *?inner*. Both template terms have variable *?interpretation* as their second argument to ensure that they are resolved under the same example-based interpretation. Otherwise, both queries would have different solutions. Section 7.3 will demonstrate that it is not always possible to eliminate the `:=` operator by composing template terms in this manner.

Non-native operator ! is a complement operator. The following template term matches classes that do not implement any interface and have a method of which the modifiers list includes a `public` modifier, but not a `static` modifier:

[4] The operator is analogous to the `@` operator of Haskell and Mercury, rather than the `:=` operator of PQL (cf. Section 3.5.3). We borrowed the syntax from the assignment operator of Smalltalk.

```
1  if jtClassDeclaration(?class){
2       class ?name implements ![?interface] {
3         public ![static] ?type::jtType ?mName(?pList) ?sList
4       }
5     }
```

The following query is equivalent and does not use the complement operator, but is less concise:

```
1  if jtClassDeclaration(?class,?interpretation){
2       class ?name {
3         public ?type::jtType ?mName(?pList) ?sList
4       }
5     },
6     absolutelyNot(jtClassDeclaration(?class,?interpretation){
7                     class ?name {
8                       static ?type::jtType ?mName(?pList) ?sList
9                     }
10                  }),
11    jtClassDeclaration(?class,?interpretation) {
12      class ?name implements ?interface ?memberList
13    },
14    [?interface isNil]
```

The first condition is a template term that contains all positive (i.e. non-complemented) concrete syntax elements of the original template term. It matches a class *?class* that has a method with a `public` modifier.

The second condition only succeeds if the same method in *?class* does not have a `static` modifier. It is equivalent to the second occurrence of the complement operator in the original template term. The higher-order predicate `absolutelyNot/n` only succeeds if the conjunction of its arguments does not succeed (cf. Section 6.2.2). The regular `not/n` predicate also succeeds if its arguments can be proven to a less than perfect truth degree.

The third and fourth condition are equivalent to the first occurrence of the complement operator in the original template term. They express that the AST node bound to *?class* should not have a child node representing an implemented interface type. The solutions to the third condition include classes that implement and classes that do not implement an interface type. In the latter solutions, *?interface* is bound to `nil`. This behavior is consistent with the matching of `return` statements with and without an expression operand (cf. Figure 4.6).

Our prototype does not support complement operators that precede instructions in a method declaration template. Under the control flow interpretation, template terms share the same limitation as the control flow traversal predicates they compile to (cf. Section 5.5.2). It is not possible to express an existential path query with a complement such as *"does there exist a path through `?m` on which anything but `b` is executed between `a()` and `c(?arg)`?"*. Section 7.4.3 demonstrates how the closely related universal path query *"is it true that there is not a single path on which `c(?arg)` follows `b()` and `b()` follows `a()`?"* can be expressed by negating a method declaration template.

***Non-native operator** ** is a reflexive transitive closure operator that can only be used in two specific places. The logic rule in the top-left corner of Figure 7.1 illustrates that the operator can also be used before the member declarations of a class (or interface) declaration template. In this position, it matches the specified members against the combined members of the class itself and all

```
1  ?class classHas: ?member under: ?interp if
2    jtClassDeclaration(?class,?interp){
3       class ?className *{ ?member }
4    }

5  ?class classHas2: ?member under: ?interp if
6    jtClassDeclaration(?class,?interp){
7      class ?className { ?member }
8    }
9  ?class classHas2: ?member under: ?interp if
10   jtClassDeclaration(?class,?interp){
11     class ?className extends ?super ?memberList
12   },
13   ?super classHas2: ?member under: ?interp
```

```
14 ?class classHas3: ?member under: ?interp if
15   jtClassDeclaration(?class,?interp){
16     class ?className extends* ?super ?memberList
17   },
18   jtClassDeclaration(?super,?interp){
19     class ?superName { ?member }
20   }
21 ?class classHas3: ?member under: ?interp if
22   jtClassDeclaration(?class,?interp){
23     class ?className { ?member }
24   }

25 ?class classHas4: ?member under: ?interp if
26   jtClassDeclaration(?class,?interp){
27     class ?className  { ?member }
28   }
29 ?class classHas4: ?member under: ?interp if
30   jtClassDeclaration(?class,?interp){
31     class ?className extends* ?superList ?mList
32   },
33   ?superList contains: ?super,
34   jtClassDeclaration(?super,?interp){
35     class ?superName { ?member }
36   }
```

Figure 7.1: Four equivalent template terms illustrating non-native syntax.

of its super classes.[5] The implemented predicate therefore quantifies over all members *?member* defined in *?class* or its super-classes. The recursive rule in the bottom-left corner illustrates how this operator can be eliminated. Its first clause stops the recursion by quantifying over all members of a class. Its second clause recurses into each super class.

The logic rules at the right of Figure 7.1 illustrate that the operator can be used after the `extends` (or `implements`) keyword of a class (or interface) declaration template. In this position, it matches the concrete syntax it precedes against the reflexive transitive closure of the super types. The `jtClassDeclaration(?c){class ?n extends* java.lang.Exception}` template term, for instance, matches all direct and indirect descendants of `java.lang.Exception`. The template term `jtClassDeclaration(?c){class ?n extends* ![java.lang.Object]}`, on the other hand, has no matches as all Java classes descend from `Object`. Likewise, variable *?super* gets bound successively to each super type of *?class* upon backtracking over the template term on lines 15–17. Variable *?superList* in the template term on lines 30–32 collects all of these super types. The rules in which both terms reside are not recursive, but implement the same predicate as the other depicted rules.

7.1.2 Semantics of Template Terms in a Nutshell

The non-native operators enumerated above conclude the discussion of the syntax for template terms. This section details how a template term is resolved and how the truth degree for such a goal is computed. Section 4.3.3 introduced both aspects of the semantics.

[5] Members defined in the interfaces it implements are not included.

Conceptually, each example-based interpretation transforms the template term into a logic rule. This conceptual rule quantifies over the AST nodes that exhibit the characteristics exemplified by the term's code excerpt —under the corresponding interpretation. When its body has identified a matching AST node, the node is unified with the first argument of the term. If present, the second argument is unified with the name of the example-based interpretation.

The truth degree for a solution to a template term is bounded by the example-based interpretation under which the solution matches the term's code excerpt. The corresponding rule is weighted by this upper bound. Resolving the term using this conceptual rule computes a resolution degree (cf. Section 6.1.2). It corresponds to the minimum of the truth degrees for the goals in the body of the rule, multiplied by its weight (i.e. the weight associated with the corresponding example-based interpretation). The truth degree for the solution is computed as the product of this resolution degree and the unification degree of the term's first argument and the matching AST node. This is consistent with the influence of unification degrees on the resolution degree for a regular goal (cf. Section 6.1.2).

Parsing an ambiguous source code excerpt produces a forest of ASTs (cf. Section 7.1.1). Conceptually, each example-based interpretation compiles each of the ASTs into a separate logic rule. Backtracking over the template term exhausts all choice points for a rule before the next rule is considered. The example-based interpretations are considered in the lexical order of the logic rules that implement their translational semantics (cf. Section 4.6.4).

From the above, it is clear that the example-based semantics of template terms are realized solely by the example-based interpretations of their source code excerpts. We discuss the standard interpretations in Section 7.2. Through a meta-interpreter, the next section further clarifies how template terms are resolved.

7.1.3 Meta-Interpreter for Fuzzy SOUL Extended With Template Terms

Figure 7.2 defines how template terms are resolved using the meta-circular evaluator methodology. The depicted clause extends the meta-interpreter for fuzzy SOUL (cf. Figure 6.1). The condition on line 4 unifies the "type" component of the template term bound to *&goal* with a variable argument compound term (cf. Section 5.1.1). For the template term `jtStatement(?s){return ?e;}`, this establishes the bindings *?functor*→`jtStatement` and *?args*→`<?s>`. Line 5 clarifies that a template term can have two arguments.

Parsing the Source Code Excerpt of a Template Term

Lines 6–8 clarify how the source code excerpt is extracted from the template term (line 6), converted to a list of tokens *?tokens* (line 7) and subsequently parsed by a Definite Clause Grammar [PW80] (cf. Section A.1). The functor *?functor* of the template term is used as the starting rule for the grammar (line 8). The DCG rules describe the concrete syntax of Java extended with logic variables and a minimum of non-native syntax. Figure 7.3 depicts the DCG rules that recognize the source code excerpt of the `return` statement template. They establish the binding *?ast*→`statement(return(expression(metaVar(?e))))`.

The DCG rules on lines 1–6 of Figure 7.3 construct the AST for the `return` statement. From the *?tokens* list, they consume the tokens that are recognized by the `jtExpressionOpt/1` rule —following a `return` keyword and followed by a semicolon

```
1  &goal isProvenToExtent: [?rdegree * ?udegree] aboveThreshold: ?threshold if
2    [&goal isKindOf: Soul.TemplateTerm],
3    !,
4    [&goal type] equals: ?functor@(?args),
5    or(?args equals: <?match>,?args equals: <?match,?interpretation>),
6    ?excerpt equals: [&goal source],
7    deify(?tokens,[QuotedCodeJavaTokenScanner new breakInTokens: ?excerpt]),
8    ?functor(?ast,?tokens,<>),

9    (?ast templateUnderInterpretation: ?quotedInterpVar
10        compilesTo: ?quotedGoals
11        forResult: ?quotedMatchVar) : ?implicationStrength,
12   ?quotedGoals quotedVariablesAsHiddenVariables: ?goals,
13   ?quotedInterpVar quotedVariablesAsHiddenVariables: ?interpVar,
14   ?quotedMatchVar quotedVariablesAsHiddenVariables: ?matchVar,

15   ?interpVar equals: ?interpretation,
16   ?goals isProvenListOfGoalsToExtent: ?rdegree
17          aboveThreshold: ?threshold
18          runningMin: 1
19          implicationStrength: ?implicationStrength,
20   ?matchVar equals: ?match : ?udgree
```

Figure 7.2: Meta-interpreter excerpt clarifying fuzzy resolution of template terms.

```
1  jtStatement(statement(?s)) --> jtStatementWithoutTrailingSubstatement(?s)
2  jtStatementWithoutTrailingSubstatement(?s) --> jtReturnStatement(?s)
3  jtReturnStatement(return(?exp)) --> <keyword([#return])>,
4                                      jtExpressionOpt(?exp),<token([#';'])>
5  jtExpressionOpt(expression(epsilon)) --> <>
6  jtExpressionOpt(?exp) --> jtExpression(?exp)

7  jtExpression(expression(?exp)) --> jtAssignmentExpression(?exp)
8  ...
9  jtPrimary(?exp) --> jtMetaVariable(?var),jtPrimaryNameRest(?var,?exp)
10 jtPrimaryNameRest(?name,?name) --> <>
11 jtPrimaryNameRest(?name,send(epsilon,?name,?args)) --> jtArguments(?args)
```

Figure 7.3: DCG rules parsing code excerpt of term `jtStatement(?s){return ?e;}`.

token (lines 3–4). The expression operand of the `return` statement is optional. The first clause of the `jtExpressionOpt/1` rule therefore succeeds without consuming any tokens (line 5). In the example term, meta-variable *?e* substitutes for the expression operand. It is recognized as a valid expression by the `jtMetaVariable/1` goal on line 9.

Line 8 of the meta-interpreter clarifies how ambiguous source code excerpts are handled. Backtracking over this starting goal for the DCG successively binds *?ast* to all possible ASTs for the excerpt. DCG rules inherently support ambiguous grammars (cf. Section A.1).

Compiling an AST for an Excerpt to Logic Goals

Lines 9–14 of Figure 7.2 clarify how an AST for the excerpt in a template term is compiled to a conjunction of logic goals. These goals correspond to one of the example-based interpretations of the excerpt. Each interpretation com-

piles the AST to a different conjunction of goals. The translational semantics are defined by logic rules that implement predicate `templateUnderInterpretation:compilesTo:forResult:/4`. This predicate is the meta-level interface through which additional example-based interpretations can be defined (cf. Section 7.5). It is called by the goal on lines 9-11. It is given the AST for the excerpt *?ast* as its input argument, while the other arguments are output arguments. The goal binds *?quotedGoals* to a list of goals that, when resolved, quantify over the program representation. The variables in this list are quoted and have to be unquoted (i.e. transformed to logic variables) before the goals can be resolved.[6] In fact, predicate `quotedVariablesAsHiddenVariables:/2` ensures that these variables are unique to each template term and hidden from its solutions (lines 12–14). This way, goals corresponding to different terms do not interfere and users only see bindings for the variables in their excerpts.

For the `jtStatement(?s){return ?e;}` template term, the solutions to lines 9–14 include bindings *?goals*→`<?Var1544669 equals: syntactic, ?Var1544668 isStatement, ?Var1544668 equals: returnStatement(?Var1544670), ?Var1544670 equals: ?e>`, *?interpVar*→`?Var1544669` and *?matchVar*→`?Var1544668`. The former variable is bound to the name of the interpretation under which the `return` statement bound to the latter variable matches the excerpt.

The meta-interpreter performs the compilation step described here at runtime. The actual evaluator for SOUL already performs this step at compile-time (cf. Section 7.5).

Quantified Resolution of the Compiled Goals

Lines 15–20 clarify how the template term bound to *&goal* is resolved and how the truth degree for this goal is computed. Conceptually, the template term is resolved using a fuzzy logic rule weighted by *?implicationStrength*. Its body consists of the goals *?goals* to which the term compiles under the example-based interpretation named *?interpVar*. Each interpretation associates a different truth degree *?implicationStrength* with the result of this compilation step (line 11). This degree functions as an upper bound for the truth degrees of the term's solutions under this interpretation. The resolution degree *?rdgree* for *&goal* corresponds to the minimum of the truth degrees for *?goals* multiplied by *?implicationStrength* (lines 16–19) — consistent with the resolution degree for a regular goal (cf. Section 6.1.2).

Resolving *?goals* establishes a binding for variable *?matchVar*: an AST node that matches the source code excerpt of the term under the example-based interpretation named *?interpVar*. Lines 20 and 15 unify variables *?interpVar* and *?matchVar* with the second and first argument of the template term respectively.[7] The truth degree for *&goal* is computed as the product of the resolution degree *?rdegree* and

[6] Quoted variables facilitate defining the translational semantics as logic rules. They can be bound to regular logic variables and passed around. In Figure 4.14, for instance, the quoted variable bound to *?baseExpression* is passed from a rule that compiles statement templates to the rule that compiles expression templates. The quoted variable represents the variable to which matching expressions will be bound at run-time.

[7] The latter unification is performed *after* the compiled goals *?Var1544668* `isStatement, ?Var1544668 equals: returnStatement(?Var1544670)` have established a binding for *?matchVar*. This way, the goal `jtStatement(?s){return ?e;}` can succeed if *?s* is already bound to any AST node that unifies with the `return`-statement matching source code excerpt.

```
1  if jtMethodDeclaration(?m,?interpretation) {
2       !static public synchronized ?type::jtType ?name(?a) {
3         3; ?i2; return ?rec.?message(?a);
4       }
5     }
```

Figure 7.4: Method declaration template term illustrating translational semantics.

the unification degree *?udegree* (line 1) —consistent with the influence of unification degrees on the resolution degree for a regular goal (cf. Section 6.1.2).

The next section details the predefined example-based interpretations that realize the example-based semantics of template terms.

7.2 Predefined Example-Based Interpretations

Template terms are resolved using multiple example-based interpretations. Section 4.3.2 introduces the example-based interpretations that are predefined in the prototype used to validate our approach. Its open-ended implementation enables users to define additional interpretations.

Conceptually, each example-based interpretation compiles the code excerpt of a template term to a logic rule. We will illustrate their translational semantics using the method declaration template term depicted in Figure 7.4. Its code excerpt corresponds to a non-`static` method that is declared `public` and `synchronized`. It has a single parameter *?a* that is used as an argument to a method invocation in its body. This invocation is the operand of a `return` statement that is preceded by two other instructions: "`3`" and "*?i2*".

The former instruction illustrates a minor syntax deviation. Java programs cannot use integers as statements. In template terms, all expressions can be used as statements. Otherwise, specifications would have to enumerate all syntactically allowed occurrences for such an expression: as the right-hand side of an assignment expression used as a statement, in the expression part of a `while`-statement, etc.

7.2.1 Syntactic Interpretation

Under the syntactic interpretation, AST nodes match a template term if they exhibit the syntactic characteristics exemplified by its code excerpt. Moreover, matching nodes should not exhibit any other syntactic characteristics. More precisely, the matching AST node and the AST for the excerpt unify under the domain-specific unification extension that unifies AST nodes with structurally equivalent compound terms —provided their abstract grammars are compatible.

Example Match The points of variation among the matches are restricted to those indicated explicitly by the logic variables in the excerpt. Therefore, the template term of Figure 7.4 has no matches under the syntactic interpretation. After all, Java programs cannot use expression "`3`" as a statement. Note that multiple occurrences of the same variable need not be bound to the same AST node under the domain-specific unification procedure.

```
1  ?Var1715032 equals: syntactic,
2  ?Var1715031 isNonConstructorMethodDeclaration,
3  ?Var1715031 equals: methodDeclaration(?,?Var1715033,?,?type,?Var1715036,
4                                        ?Var1715034,?,?,?Var1715035),
5  ?name equals: ?Var1715036,
6  [1] isSizeOf: ?Var1715034,
7  ?Var1715034 equals: nodeList(<?Var1715052>),
8  ?a equals: ?Var1715052,

9  [2] isSizeOf: ?Var1715033,
10 ?Var1715033 equals: nodeList(<?Var1735803,?Var1735804>),
11 modifier(['public']) equals: ?Var1735803,
12 modifier(['synchronized']) equals: ?Var1735804,
13 absolutelyNot@(<?Var1715033 contains: ?Var17358020,
14                 modifier(['static']) equals: ?Var1735802>),

15 ?Var1715035 equals: block(?Var1715043),
16 [3] isSizeOf: ?Var1715043,
17 ?Var1715043 equals: nodeList(<?Var1715044,?Var1715046,?Var1715047>),

18 ?Var1715044 equals: expressionStatement(?Var1715045),
19 numberLiteral(['3']) equals: ?Var1715045,

20 ?i2 equals: ?Var1715046,

21 ?Var1715047 equals: returnStatement(?Var1715048),
22 ?Var1715048 equals: methodInvocation(?Var1715049,?,?message,?Var1715050),
23 ?Var1715049 equals: ?rec,
24 [1] isSizeOf: ?Var1715050,
25 ?Var1715050 equals: nodeList(<?Var1715051>),
26 ?Var1715051 equals: ?a
```

Figure 7.5: Syntactic interpretation of the template term in Figure 7.4.

Corresponding Goals Figure 7.5 depicts the goals that are used to resolve the template term depicted in Figure 7.4. These goals quantify over the AST nodes that exhibit the syntactic characteristics exemplified by the term's code excerpt. They express these characteristics through the predicates of the CAVA library that reify syntactic information (cf. Section 5.2.1). The binding for variable *?Var1715031* (line 2) is a matching AST node. It will be unified with the first argument of the template term (cf. Section 7.1.2). The binding for variable *?Var1715032* (line 1) is the name of the syntactic interpretation. It will be unified with the second argument of the term. Variables of this form are unique to the term and hidden from its solutions (cf. Section 7.1.3).

Line 3 unifies the method declaration AST node with a structurally equivalent compound term to access its children (cf. Section 6.4.2). Line 5 binds the "name" child to variable *?name* in the code excerpt. Lines 6–8 bind variable *?a* to the single formal parameter of the method declaration. These lines are the same under all predefined interpretations.

Lines 9–14 express the syntactic characteristics of the "modifiers" child bound to *?Var1715033*. This `ASTNode$NodeList` instance has to contain two modifiers in the order exemplified by the template term (lines 10–12). No other modifiers are allowed. Lines 13–14 correspond to the complement operator before the `static` modifier in the template term. They are redundant as the preceding lines have already restricted the size of the modifiers list. This redundancy is due to the generic manner in which template lists are compiled (cf. Section 7.5).

Lines 15–17 express that the "body" child of the method declaration has to be a block that wraps an `ASTNode$NodeList` instance containing three instructions.

Again, these have to occur in the order exemplified by the template term. No other instructions are allowed.

Lines 18–19 restrict the first instruction in the method to expression "3" used as a statement. As this is impossible in Java, the template term does not have any solutions under the syntactic interpretation. Line 20 unifies the second instruction with variable *?i2* of the code excerpt. Lines 21–26 restrict the third instruction to a return statement that has the exemplified method invocation as its operand.

Points of variation among the matches for the term are restricted to the explicit points of variation indicated by the logic variables in its excerpt. In fact, lines 3–26 *could* be replaced by a single condition that unifies the AST for the excerpt with the method declaration node *?var1715031* if their abstract grammars *were* compatible. This would rely on the domain-specific unification extension that unifies AST nodes with structurally equivalent compound terms (cf. Section 6.4.2).

7.2.2 Lexical Interpretation

The lexical interpretation is a less restrictive version of the syntactic interpretation. Matches have to exhibit the syntactic characteristics exemplified by the excerpt, but are allowed to exhibit additional ones. This accounts for implicit variation points. However, the lexical relations among the elements have to be the same as the ones among the corresponding elements in the excerpt. If a statement in the excerpt is preceded by a local variable declaration, for instance, matching statements have to be preceded by a matching variable declaration as well.

Example Match The following method matches the template term of Figure 7.4 under the lexical interpretation:

```
1 synchronized public Integer callsite_2(Integer arg_2){
2   if (5 > 3) { return called_from_multiple_sites(arg_2); }
3   else return callsite_1(arg_2);
4 }
```

Note that it features all exemplified modifiers, but not in the same order. Its body also features all exemplified instructions, but they are nested within other instructions. We explain the bindings for instruction *?i2* within this match below.

Corresponding Goals Figure 7.6 depicts the goals that are used to resolve the template term. The first 8 lines are similar under all example-based interpretations.

Under the syntactic interpretation, matching methods were required to have exactly two modifiers in the exemplified order (lines 9–14 of Figure 7.5). Under the lexical interpretation, matching methods are allowed to have additional modifiers. There is no restriction on their order. Variable *?Var1715053* is bound to the `ASTNode$NodeList` instance that contains the modifiers of the matching method. There should be at least two (lines 9–10). Predicate `collectionContains:andAlso:/3` is defined in the standard library of SOUL. It unifies its second argument with an element chosen from the collection that is given as its first argument. The remaining elements are unified with its third argument. Line 11 therefore extracts a `public` modifier from the modifiers collection and line 13 extracts a `synchronized` modifier from the remaining modifiers. Upon backtracking, both lines quantify over all combinations of a `public` and `synchronized` modifier in this collection (i.e. without repetition). Lines 15–16 ensure that the collection

```
1  ?Var1715032 equals: lexical,
2  ?Var1715031 isNonConstructorMethodDeclaration,
3  ?Var1715031 equals: methodDeclaration(?,?Var1715053,?,?type,?Var1715056,
4                                        ?Var1715054,?,?,?Var1715055),
5  ?name equals: ?Var1715056,
6  [1] isSizeOf: ?Var1715054,
7  ?Var1715054 equals: nodeList(<?Var1715071>),
8  ?a equals: ?Var1715071,

9  ?Var1715059 isSizeOf: ?Var1715053,
10 ?Var1715059 isEqualToOrGreaterThanButRelativelyCloseTo: [2],
11 ?Var1715053 collectionContains: ?Var1715060 andAlso: ?Var1715061,
12 modifier(['public']) equals: ?Var1715060,
13 ?Var1715061 collectionContains: ?Var1715062 andAlso: ?Var1715063,
14 modifier(['synchronized']) equals: ?Var1715062,
15 absolutelyNot@(<?Var1715053 contains: ?Var1715058,
16                 modifier(['static']) equals: ?Var1715058>),

17 ?Var1715055 blockIsLexicalCandidateForAmountOfActualStatements: [1],

18 ?Var1715064 isStatementOrExpressionInScopeOf: ?Var1715055,
19 numberLiteral(['3']) equals: ?Var1715064,

20 ?Var1715065 isStatementOrExpressionInScopeOf: ?Var1715055,
21 ?Var1715065 followsASTNode: ?Var1715064,
22 ?i2 equals: ?Var1715065,

23 ?Var1715066 isStatementInScopeOf: ?Var1715055,
24 ?Var1715066 followsASTNode: ?Var1715065,
25 ?Var1715066 equals: returnStatement(?Var1715067),
26 ?Var1715067 equals: methodInvocation(?Var1715068,?,?message,?Var1715069),
27 ?Var1715068 equals: ?rec,
28 [1] isSizeOf: ?Var1715069,
29 ?Var1715069 equals: nodeList(<?Var1715070>),
30 ?Var1715070 equals: ?a
```

Figure 7.6: Lexical interpretation of the template term in Figure 7.4.

does not contain a `static` modifier. The goal uses the higher-order predicate `absolutelyNot/n` in variable argument notation (cf. Section 5.1.1).

The body of the method declaration template in Figure 7.4 exemplifies three instructions. The first instruction is an expression that is used as a statement. Matching method declaration AST nodes are required to feature a corresponding expression or statement at any level of nesting (lines 18–19). In method `callsite_2`, for instance, the corresponding expression *?Var1715064* is nested within the expression child of an `if`-statement. Predicate `isStatementOrExpressionInScopeOf:/2` quantifies over all expressions and statements at any level of nesting within its second argument.

Variable *?i2* is the second instruction in the method declaration template of Figure 7.4. It can stand for any statement or expression at any level of nesting within a matching method declaration AST node. However, a lexical ordering constraint (i.e. based on line numbers) is imposed on the instructions within this node. Matches for the first instruction (bound to *?Var1715064*) lexically precede the matches for the second instruction (bound to *?Var1715065*). Line 21 asserts this through predicate `followsASTNode:/2` (cf. Section 5.3.3). Method `callsite_2` therefore matches the template term with the following bindings for *?i2*: the block "`{return called_from_multiple_sites(arg_2);}`", the statement "`return called_from_multiple_sites(arg_2);`", the expression "`called_from_multiple_sites(arg_2)`" and the expression "`arg_2`". All of these bindings lexically precede statement "`return callsite_1(arg_2);`"

which matches the third template instruction "`return ?rec.?message(?a);`".

The third template instruction is an actual statement (i.e. not an expression used as a statement): a `return`-statement with a method invocation as its operand. Lines 23–30 ensure that matching method declaration AST nodes feature a corresponding statement *?Var1715066* at any level of nesting (line 23). This statement should lexically follow the node corresponding to the second instruction in the template (line 24). Method `callsite_2` matches the template term under the lexical interpretation with *?message* bound to "`callsite_1`".

7.2.3 Control Flow Interpretation

Most template terms are resolved under the control flow interpretation as if they were resolved under the lexical interpretation. Only method declaration templates and method declarations within type declaration templates are resolved differently.

Under the control flow interpretation, matching method declarations have to exhibit the control flow characteristics exemplified by the template term. There should be a path through the control flow graph of the method (i.e. existentially qualified) on which all exemplified instructions are executed. Implicit variation points are supported. Non-specified instructions are allowed on the execution path. The path also crosses method boundaries (i.e. it is inter-procedural). As a result, matches for an instruction in the template term need not reside in the method declaration that matches the term. They can also reside in a method called (directly or transitively) by the matching method declaration. However, actual statements such as `return`-statements are matched intra-procedurally.

Example Match Method `callsite_2` (cf. Section 7.2.2) matches the template term of Figure 7.4 under the control flow interpretation. However, the bindings for the logic variables within the term differ from the previous section. Variable *?message*, for instance, can be bound to "`callsite_1`" (in the `true`-branch of the `if`-statement) and to "`called_from_multiple_sites`" (in the `false` branch). Both bindings reside on different paths through the control flow graph of the method. The corresponding bindings for variable *?i2* differ as well.

Corresponding Goals Figure 7.7 depicts the goals that are used to resolve the template term. These goals quantify over the method declarations that exhibit the exemplified control flow characteristics. They express these characteristics through the predicates of the CAVA library that reify control flow information (cf. Section 5.2.1). Lines 1–15 are the same as under the lexical interpretation.

Actual statements (i.e. not expressions that are used as a statement) are matched intra-procedurally. This is illustrated for the `return`-statement in the template by the goals on lines 18 and lines 33–34 respectively. The former goal asserts that the matching AST node (bound to *?Var1715552*) lexically resides within the matching method declaration. The latter goal ensures that this statement is executed after the nodes matching "*?i2*" and "3" have been executed. These AST nodes (bound to *?Var1715557* and *?Var1715558*) are found on the inter-procedural execution path by lines 29–32 and lines 26–28 respectively.

Line 21 requires an AST node that matches the operand `?rec.?message(?a)` (i.e. *?Var1715553*) of the specified `return`-statement to be executed before the AST node that matches said statement (i.e. *?Var1715552*), but after the AST node that

```
1  ?Var1715500 equals: controlflow,
2  ?Var1715499 isNonConstructorMethodDeclaration,
3  ?Var1715499 equals: methodDeclaration(?,?Var1715540,?,?type,?Var1715543,
4                                        ?Var1715541,?,?,?Var1715542),
5  ?name equals: ?Var1715543,
6  ?Var1715541 equals: nodeList(<?Var1715559>),
7  ?a equals: ?Var1715559,

8  ?Var1715546 isSizeOf: ?Var1715540,
9  ?Var1715546 isEqualToOrGreaterThanButRelativelyCloseTo: [2],
10 ?Var1715540 collectionContains: ?Var1715547 andAlso: ?Var1715548,
11 modifier(['public']) equals: ?Var1715547,
12 ?Var1715548 collectionContains: ?Var1715549 andAlso: ?Var1715550,
13 modifier(['synchronized']) equals: ?Var1715549,
14 absolutelyNot@(<?Var1715540 contains: ?Var1715545,
15                modifier(['static']) equals: ?Var1715545>),

16 ?Var1715551 isControlFlowTraversalState,
17 ?Var1715542 blockIsLexicalCandidateForAmountOfActualStatements: [1],

18 ?Var1715552 isStatementInScopeOf: ?Var1715542,

19 ?Var1715555 equals: nodeList(<?Var1715556>),
20 ?Var1715553 equals: methodInvocation(?Var1715554,?,?message,?Var1715555),
21 ?Var1715553 inFlowOf: ?Var1715542 following: <?Var1715556>
22                                    before: <?Var1715552>,
23 ?a equals: ?Var1715556,
24 ?rec equals: ?Var1715554,
25 ?Var1715552 equals: returnStatement(?Var1715553),

26 numberLiteral(['3']) equals: ?Var1715557,
27 ?Var1715557 inFlowOf: ?Var1715542 following: <>
28                                     before: <?Var1715558,?Var1715552>,

29 ?i2 equals: ?Var1715558,
30 ?Var1715558 inFlowOf: ?Var1715542 following: <?Var1715557>
31                                    before: <?Var1715552>,
32 ?Var1715558 isGroundStatementOrExpression,

33 ?Var1715552 inFlowOf: ?Var1715542 following: <?Var1715558,?Var1715557>
34                                    before: <>
```

Figure 7.7: Control flow interpretation of the template term in Figure 7.4.

matches the argument of the invocation (i.e. *?Var1715556*). This corresponds to the order in which the children of such a `return`-statement would be executed at run-time. Note that the matching expression *?Var1715553* is not necessarily the "operand" child of the `return`-statement. Line 25 merely requires both AST nodes to unify according to the domain-specific unification procedure. As a result, the following method declaration template also matches the template of Figure 7.4:

```
1 public synchronized Integer othermethod(Integer arg) {
2   Object temp = new Integer(3);
3   temp = this.callsite_2(arg);
4   return (Integer) temp;
5 }
```

7.2.4 Shared Implicit Variation Points

The points of variation among the matches for a template differ under each interpretation. This was explained by the previous sections. However, all interpretations recognize different implementations of the same data flow characteristic (i.e. implicit variation points). In template terms, data flow characteristics are expressed through multiple occurrences of the same variable. Their bindings have to unify

according to the domain-specific unification procedure which is the same for all interpretations.

Shared support for other implicit variation points is due to a common design principle. Matches for a template term have to exhibit all exemplified characteristics.[8] What is not exemplified cannot constrain these matches further —except under the syntactic interpretation which requires exact matches. Under the lexical and control flow interpretation, the following template and field declaration therefore match —successively binding *?field* to `i` and `j` upon backtracking:

```
1  jtFieldDeclaration(?dec){ public int ?field = ?init; }
2  public int i=0, j=1;
```

Likewise, the following template and method declaration match. The latter's variable declaration with an initializer expression can be considered shorthand for an initializer-free variable declaration followed by an assignment:

```
1  jtMethodDeclaration(?m){ void m() { ?x = ?init; } }
2  void m() { int x = 3; }
```

However, the converse is not true. A method declaration template with a local variable declaration requires its matches to declare a local variable.

Support for such implicit variation points is not implemented in the domain-specific unification procedure, but in the translational semantics of the example-based interpretations. SOUL does not support choice points in its unification procedure. This would be required for the field declaration example above. Moreover, the abstract grammars for the base program and code excerpts are not structurally compatible. This precludes unifying a base program AST node with a compound term that represents the AST for the source code excerpt.

7.2.5 Truth Degrees for Example-Based Interpretations

In our approach, pattern detection results are ranked according to the extent to which they exhibit the characteristics in a specification. The smaller this extent, the more likely a reported instance is a false positive (cf. Section 4.5.2).

Concretely, the fuzzy logic cornerstone associates truth degrees with each solution to a query (cf. Section 6.1). The truth degrees for solutions to a template term are bounded by the example-based interpretation under which the term is resolved. Truth degrees for solutions identified under the syntactic, lexical and control flow interpretation can be no larger than 1, $\frac{9}{10}$ and $\frac{8}{10}$ respectively. This ranking reflects the projected similarity of the solutions to the code excerpt of the term. Under the control flow interpretation, for instance, matches for an instruction in a method declaration template need not reside in the method declaration

[8] Exceptions to this design principle are few. Template `jtExpression(?e){this.?field}`, for instance, also matches field accesses with an implicit base expression. This is because such field accesses cannot be quantified over using a `jtExpression(?e){?field}` template —at least, not without suffixing *?field* with the `::jtFieldAccess` parser directive. Template `jtClassDeclaration(?class){class ?name implements ?interface ?memberList}` also matches class declarations that do not implement an interface type —binding *?interface* to `nil`. Otherwise, a disjunction of two templates would have to be used to quantify over all classes. Likewise, templates `jtStatement(?statement){return ?e;}` and `jtFieldDeclaration(?dec){?modList ?type ?field = ?init;}` quantify over all `return`-statements and field declarations —regardless of whether they have an expression operand and initializer expression respectively.

that matches the term (cf. Section 7.2.3). Conceptually, each interpretation compiles the term to a fuzzy logic rule that is weighted by this upper bound. The term is resolved using this rule (cf. Section 7.1.2).

The properties of the solution itself further refine this upper bound. Solutions are ranked lower if they required a domain-specific unification that could introduce false positives (cf. Section 6.4). For instance, if the bindings for two occurrences of the same variable are in a may-alias relation. Solutions are also ranked lower if they exhibit more characteristics than are exemplified by the template —except under the syntactic interpretation, which requires exact matches. For instance, if a method declaration AST node has more modifiers than the ones that are exemplified. This is ensured by goals involving predicate `isEqualToOrGreaterThanButRelativelyCloseTo:/2` (e.g. line 10 and line 9 in Figure 7.6 and Figure 7.7 respectively). They succeed with a truth degree of $\frac{9}{10}$ for all numbers *?x* that are greater than *?y*, but deviate more than 10% from *?y* (cf. Section 6.2.1).

This concludes our discourse on the standard example-based interpretations for template terms. Section 7.4 further demonstrates their practical differences on realistic example-based specifications.

7.3 Composing Template Terms

Template terms can be used anywhere a regular logic term is allowed. This enables composing example-based specifications from template terms and logic connectives such as `and/n`, `or/n` and `not/n`. Section 7.1.1 makes extensive use of this feature to eliminate non-native syntax from the source code excerpts of template terms. In this section, we will illustrate that composing terms in such a manner allows for fine-grained control over their matches.

Figure 7.8 depicts two queries that are resolved under the control flow interpretation (cf. Section 7.2.3). The depicted queries are closely related in the sense that both feature the same concrete syntax elements. Their solutions differ because these elements have been divided over different template terms —effectively controlling the semantics of the resulting example-based specification in a fine-grained manner:

- The query at the top of Figure 7.8 matches methods that lexically feature a `return`-statement in their body. Preceding this statement, matching methods furthermore have an expression matching *?x*`.m()` on a path through their control flow graph. This expression unifies with the "operand" child of the `return`-statement. As there is no such method at the right of the figure, this query has no solutions.

- The closely related query at the bottom of Figure 7.8, in contrast, has two solutions. It has two template terms. The first template term quantifies over all expressions *?e1* that match *?x*`.m()`: expression `x.m()` in method `methodC()`. The second template term quantifies over all method declarations that lexically feature a `return`-statement with operand *?e2*: methods `m` and `methodM` with operands `new Integer(111)` and `o.f` respectively. The query succeeds because `methodC()` establishes their may-alias relation with expression `x.m()`.

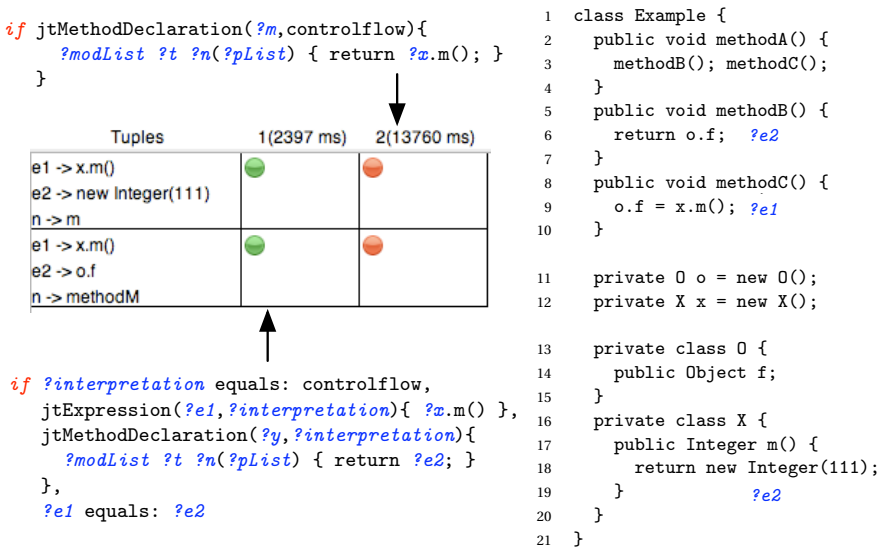


Figure 7.8: Fine-grained control over matches through template composition.

7.4 Revisiting LMP Support for Pattern Characteristics

Having instantiated the example-based specification cornerstone, we demonstrate how it improves upon the support offered by LMP for each kind of pattern characteristic (cf. Section 5.3).

7.4.1 Expressing Syntactic Characteristics

Section 5.3.1 introduces the syntactic characteristics of potentially enhanceable `for`-statements. Figure 5.6 depicts two example-based specifications that identify such statements.

The first query consists of a template term for a Java statement *?s* (lines 1–4) and a logic condition that queries *?s* for its parent method declaration *?m* (line 5). Bindings for *?s* are AST nodes that match the concrete syntax in the template term under one or more example-based interpretations. The second query is equivalent to the first query. It consists of a single template term for a Java method declaration *?m* that contains a potentially enhanceable for-statement *?s*.

The source code excerpt in each template term corresponds to the prototypical implementation of the pattern's *essential* characteristics only. For instance, they do not require `for`-statements to have a block as their body. The following template term, in contrast, would because of the extra braces on lines 2 and 4:

```
1  if jtStatement(?s){
2      for(?initList; ?hasNextReceiver.hasNext(); ?updList) {
3        ?nextReceiver.next();
4      }
5    }
```

The example-based queries identify the same solutions as the regular LMP query (cf. Figure 5.6) and the LMP query that relies on domain-specific unifica-

```
1  if jtStatement(?s){
2       for(?initList; ?hasNextReceiver.hasNext(); ?updList)
3         ?nextReceiver.next();
4     },
5     [?s parentMethodDeclaration] equals: ?m

6  if jtMethodDeclaration(?m){
7       ?modList ?type ?name(?paramList) {
8         ?s := for(?initList; ?hasNextReceiver.hasNext(); ?updList)
9                 ?nextReceiver.next();
10      }
11    }
```

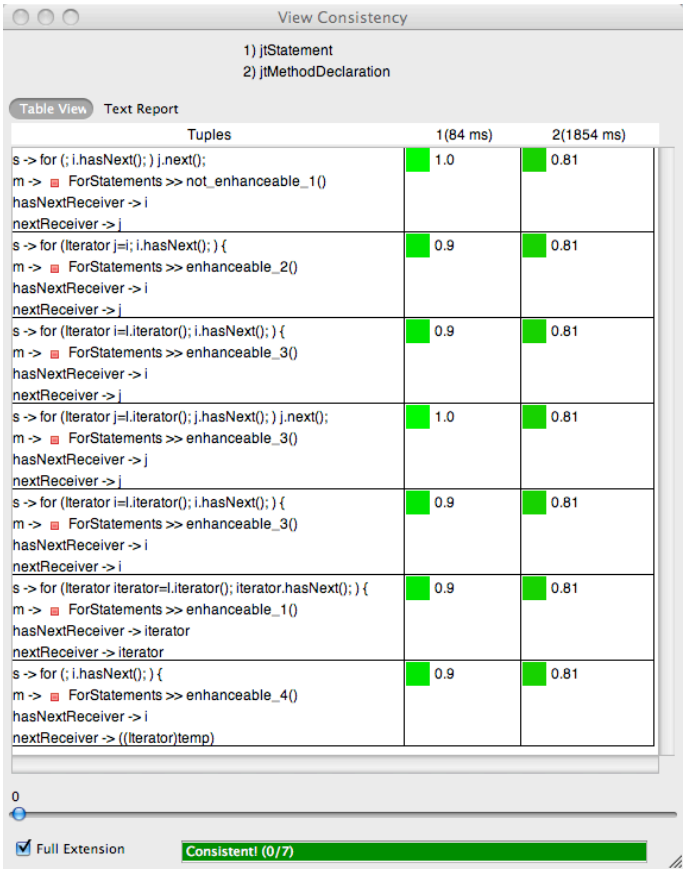


Figure 7.9: Example-based spec. for syntactic char. of enhanceable `fors`.

tion (cf. Figure 6.8).[9] They originate from the program depicted in Figure 5.6. The quantified solutions for the example-based queries are depicted at the bottom of Figure 7.9. Each row corresponds to a solution. The first column depicts the bindings for variables *?s*, *?m*, *?hasNextReceiver* and *?nextReceiver* in each solution. The second and third column depict the *maximum* truth degree associated with these bindings in the solutions to the first and second query respectively.[10] A truth degree of 0 in one of these columns would indicate that the corresponding query did not identify a solution identified by the other query. As there are no such entries, both queries identify the same solutions. However, the queries associate different truth degrees with each solution. Below we explain that this is because the first query exemplifies potentially enhanceable `for`-statements, while the second query exemplifies methods in which such a statement resides.

Truth degrees for solutions to the first query Solutions with a maximum truth degree of 1 match the template term under the syntactic interpretation. They do not exhibit any other characteristics than those that are exemplified by the template term. The first column does have entries with a maximum truth degree of 1. The inner `for`-statement in method `enhanceable_3()` and the `for`-statement in method `not_enhanceable_1()` are perfect matches for the statement template in the first query. The points of variation among these statements are restricted to the explicit points of variation indicated in the template term (i.e. its logic variables).

Solutions with a maximum truth degree of 0.9 match the template term under the lexical interpretation. This is the case for all other `for`-statements identified by the first query. They exhibit all of the syntactic characteristics exemplified by the template term, but their body is a block statement in which the invocation of method `next()` resides rather than the invocation itself. This is allowed under the lexical interpretation as long as the lexical relations among the AST nodes in a match are the same as the ones among the corresponding concrete syntax elements in the template term. The `next()` invocation therefore only has to reside lexically in the body of the `for`-statement.

Truth degrees for solutions to the second query The second column does not have an entry with a maximum truth degree of 1. None of the methods depicted in Figure 5.6 match the method declaration template under the syntactic interpretation. Each method has more statements than just the potentially enhanceable `for`-statement.

The truth degree for a solution to a template term is bounded by the example-based interpretation under which the solution matches the term. Conceptually, each example-based interpretation compiles the term to a fuzzy logic rule that is weighted by this upper bound. The term is resolved using this rule (cf. Section 6.1.2 for fuzzy resolution). The truth degree for the conjunction of the goals in its body can therefore lower the upper bound. For instance, if a solution exhibits more char-

[9] The LMP queries allow the invocation of method `next()` to reside either in the body of the `for`-statement or in one of its updater expressions. The example-based queries restrict this invocation to the body of the statement. This suffices for all statements in Figure 5.6.

[10] All example-based interpretations of a template term are considered transparently. The same solution can therefore be identified by multiple example-based interpretations —each with a different upper bound on the truth degrees for their matches. The fuzzy SOUL prototype does not aggregate identical solutions with different truth degrees into a single solution with an aggregated truth degree (cf. Section 6.1.2). We performed such an aggregation step in the depicted table to improve its readability.

```
1  ?member isNativeATMethodDefinedIn: ?interface if
2    or(jtInterfaceDeclaration(?interface,lexical){
3         interface ?interfaceName extends* ATObject { ?member }
4       },
5       jtInterfaceDeclaration(?interface,lexical){
6         interface ATObject  { ?member }
7       }),
8    ?member methodDeclarationHasName: {(meta|base)_.+}
```

Figure 7.10: Example-based equivalent for the rule in Figure 5.7.

acteristics than exemplified by the template term. This is the case for the solutions to the second query. Their truth degree of 0.81 is the product of the upper bound for the lexical interpretation (i.e. 0.9) and the truth degree for a goal *?x* `isEqualToOrGreaterThanButRelativelyCloseTo:` *?y*. Here, *?x* and *?y* are bound to the number of statements in a method declaration AST node and the number of statements in the template term respectively.[11]

Evaluation Figure 5.6 depicts a regular LMP query for potentially enhanceable `for`-statements. Its convoluted sequences of "`... has ...`" conditions evidence the unification-related shortcomings of regular LMP (cf. Section 4.4.2). The equivalent LMP query depicted in Figure 6.8 is more concise. It relies on the domain-specific unification of AST nodes with structurally equivalent compound terms. Both queries express the pattern's syntactic characteristics by quantifying over the ASTs in the program representation. This exposes users to the abstract grammar of the *abstract syntax* trees, their implementation details and their reification. Both queries evidence the quantification-related shortcomings of LMP (cf. Section 4.2.2).

The example-based specifications in Figure 7.9, in contrast, only expose users to the *concrete syntax* of the base program —augmented with a minimum of non-native syntax to indicate points of variation.

7.4.2 Expressing Structural Characteristics

In this section, we present example-based specifications for the application-specific predicates and coding conventions introduced in Section 5.3.2.

The `isNativeATMethodDefinedIn:/2` predicate revisited

Figure 7.10 depicts an example-based rule that is equivalent to the third rule at the top of Figure 5.7. Both rules implement predicate `isNativeATMethodDefinedIn:/2` which quantifies over method declarations defined in the `ATObject` interface hierarchy of which the name starts with prefix `base_` or `meta_`. The latter is checked by the second condition of the example-based rule. It relies on the domain-specific extension that unifies an AST node (i.e. the "name" part of a method declaration *?member*) with a regular expression that matches the concrete syntax of the node (cf. Section 6.4.2). The first condition in the rule is a disjunction

[11] At all levels of nesting, but excluding expressions that are used as a statement. Among the solutions depicted in Figure 7.9, bindings for *?x* range from 2 (method `enhanceable_1()`) to 4 (methods method `enhanceable_3()` and `enhanceable_4()`), while *?y* is always bound to 1.

```
1  if ?interpretation equals: lexical,
2     jtClassDeclaration(?class,?interpretation){
3       class ?cName extends* edu.vub.at.objects.natives.NativeATObject {
4         ?m := ?modList ?type ?mName(?pList) ?mStatList
5       }
6     },
7     ?mName equals: {(meta|base)_.+},

8     not(jtClassDeclaration(?class,?interpretation){
9           class ?cName extends* ?super ?decList
10        },
11        jtClassDeclaration(?super,?interpretation){
12          class ?sName {
13            ?superModList ?type ?mName(?pList) ?overriddenStatList
14          }
15        }),

16    not(or(jtClassDeclaration(?class,?interpretation){
17             class ?cName implements ?interface ?decList
18           },
19           and(jtClassDeclaration(?class,?interpretation){
20                 class ?cName extends* ?otherSuper ?decList
21               },
22               jtClassDeclaration(?otherSuper,?interpretation){
23                 class ?superName implements ?interface ?otherSuperDecList
24               })),
25        jtInterfaceDeclaration(?interface,?interpretation){
26          interface ?iName extends* ATObject {
27            ?interfaceModList ?type ?mName(?pList);
28          }
29        })
```

Figure 7.11: Example-based equivalent for the query in Figure 5.8.

of two `jtInterfaceDeclaration/2` template terms. The terms quantify over all member declarations *?member* in interface `ATObject` (second template term) or one of its sub-types (first template term). The first term has a non-native suffix `*` after its `extends` keyword. This ensures that *?interface* quantifies over the transitive closure of the "extends-`ATObject`" relation. Upon backtracking over the term, *?interface* is therefore successively bound to each sub-type of the `ATObject` interface. The original rule used the application-specific predicate `isATObjectInterface/1` to quantify over the interface hierarchy.

Note that both template terms should be resolved under the lexical interpretation because the syntactic interpretation only identifies interfaces that have a single member declaration and because the template does not exemplify any control flow characteristics.

The AMBIENTTALK coding convention revisited

The example-based query depicted in Figure 7.11 is equivalent to the query depicted at the top of Figure 5.8.[12] Both queries detect violations against the AMBI-

[12] Provided the example-based query is evaluated in regular SOUL. To evaluate the query in fuzzy SOUL, all occurrences of the `not/n` connective should be replaced by an `absolutelyNot/n` connective (cf. Section 6.2.2). These particular `not/n` occurrences should also fail when their arguments succeed with a less than perfect truth degree. We will demonstrate this in the next section.

ENTTALK coding convention introduced in Section 5.3.2. The coding convention requires that classes which define their own native methods (i.e. methods with the `base_` or `meta_` prefix that are not defined in a super class) should implement an interface in which those methods are defined. It can also be the case that the interface is implemented by a super class.

As in the previous rule, we resolve all template terms of the query under the lexical interpretation. Lines 2–7 of the example-based query are equivalent to lines 1–8 of the LMP query. They bind *?class* to a sub-type of class `NativeATObject` and *?m* to one of the native methods in this class. The `jtClassDeclaration/2` template term relies on the non-native suffix `*` after its `extends` keyword for the former (line 3) and on the non-native operator `:=` for the latter (line 4). Note that method *?m* can either be abstract or concrete. In case the method is abstract, variable *?mStatList* will be bound to `nil`. An abstract method has no body. In case the method is concrete, *?mStatList* will be bound to the `ASTNode$NodeList` instance that represents the statements in its body. Variables *?pList* and *?modList* stand for the modifier list and formal parameter list of method *?m*.

Lines 8–15 of the example-based query are equivalent to lines 9–11 of the original query. They negate (an implicit conjunction of) two template terms to express that method *?m* should not override a method from a super class *?super*. The first template term quantifies over all super classes of *?class* (and their member declarations *?decList*). The second template term matches one of these super classes if it has a method with the return type *?type*, name *?mName* and parameter list *?pList* of method *?m*.

Lines 16–24 of the example-based query are equivalent to lines 12–15 of the LMP query. They identify an interface *?interface* that is either implemented directly by class *?class* (the template term on lines 16–18) or is implemented by one of its super classes *?otherSuper* (lines 19–24). Lines 25–29 (equivalent to lines 16–18 of the LMP query) require this interface to be in the `ATObject` interface hierarchy (which parallels the `NativeATObject` class hierarchy) and to define the abstract version of method *?m*. Lines 16–24 are negated to find violations of the coding convention.

Evaluation There is little to improve upon the LMP specifications for the AMBIENTTALK coding convention introduced in Section 5.3.2. The relational nature of logic programming facilitates quantifying over the reification predicates for structural information to express their structural characteristics (cf. Section 4.2.2).

The above example-based specification exemplifies the overriding relation between two methods in terms of the inheritance relation between their classes and the syntactic characteristics of the methods. This is possible because of the domain-specific unification procedure according to which different parameter lists, individual parameters, parameter types and return types of overriding methods unify (cf. Section 6.6.2). For instance, the return types of overriding methods are allowed to be co-variant and parameters are allowed to have different names. The original LMP query quantifies over the overriding relation between two methods using reification predicate `overrides:/2` (cf. Figure 5.12).

The LMP and example-based queries identify the same violations in 25.890ms and 124.893ms respectively. Clearly, quantifying directly over the reified structural relations is less costly than exemplifying these relations. The LMP query is also more concise (at least in terms of lines of code). However, it requires knowledge

about the reification predicates for structural information (e.g. `overrides:/2`, `definesMethod:/2`, `implementsType:/2` and `inClassHierarchyOfType:/2`).

7.4.3 Expressing Control Flow Characteristics

The example-based query depicted at the top of Figure 7.12 identifies methods that comply with the control flow characteristics of the protocol introduced in Section 5.3.3. These require an invocation of method `a()` to be followed by an invocation of method `c(Object)`, without method `b()` being invoked in between.

The first condition of the query is a `jtClassDeclaration/2` template term. Upon resolution, its first argument *?class* will be unified with a class declaration AST node that matches its source code excerpt (on lines 2–8) under the example-based interpretation named *?interpretation* (its second argument). The source code excerpt exemplifies a class named `ProtocolExample` that defines a method *?m* which is not declared `static` and invokes methods `a()` and `c(Object)`. The term establishes bindings for these invocations (i.e. *?a* and *?c*) and for the type, name and parameters of the method (i.e. *?type*, *?name* and *?paramList*).

The second condition of the query verifies that method *?m* does not have an invocation of method `b()` in between the invocations of `a()` and `c(Object)`. The argument to the higher-order predicate `absolutelyNot/n` is a `jtMethodDeclaration/2` template term. Its source code excerpt exemplifies such a violation of the protocol by invoking `b()` in between the invocations of `a()` and `c(Object)`. Predicate `absolutelyNot/n` is defined in the standard library of fuzzy SOUL and succeeds only if the conjunction of its arguments fails (cf. Section 6.2.2). The fuzzy version of the regular `not/n` connective would have also succeeded if its arguments succeed with a truth degree smaller than 1.

The final two conditions are optional. We added these conditions to ensure that variables *?mods* and *?block* have the same bindings as in the original LMP query. The same goes for the occurrences of the non-native `:=` operator. The occurrence on line 4, for instance, binds *?a* to method invocation `a()`. We could have omitted these occurrences and only kept their right-hand side.[13]

The second column of the window in Figure 7.12 lists the truth degrees for the solutions to the query. The bindings for *?interpretation* clarify whether a solution was identified under the lexical or control flow interpretation of the template terms.

Solutions under the lexical interpretation Under the lexical interpretation, the example-based query identifies the same methods as the LMP query in the top-left corner of Figure 5.10 (cf. third column of the top-left window in Figure 5.9). The LMP query is roughly equivalent to the goals that are used to resolve the template terms under the lexical interpretation. We therefore refer to Section 5.3.3 for an in-depth discussion of their solutions.

Clearly, we intended the template terms to exemplify the control flow characteristics of complying methods. An additional condition `?interpretation equals: controlflow` would exclude the other interpretations from being considered.

[13] Except for the `:=` operator on line 3. It connects the first condition with the second condition through variable *?m*. It can only be omitted by substituting a `jtMethodDeclaration/2` term for the `jtClassDeclaration/2` term and adding an additional condition `[?m parentTypeDeclaration getName] equals: simpleName(['ProtocolExample'])`.

Solutions under the control flow interpretation Under the control flow interpretation, the example-based query identifies the same methods as the LMP query in the bottom-left corner of Figure 5.10 (cf. second column of the top-left window in Figure 5.9). We refer to Section 5.3.3 for a more in-depth discussion of their solutions.

The first condition of the example-based query roughly corresponds to lines 6–10 of the LMP query. Through subsequent traversals of the control flow graph of complying methods, they express the existential path query *"does there exist a path through `?m` on which `c(?arg)` follows `a()`?"*. This explains why method `semi_compliant_2` is recognized as a complying method although there is a path through the method on which `c` is never executed. The second condition corresponds to lines 11–15 of the LMP query. They express the universal path query *"is it true that there is not a single path on which `c(?arg)` follows `b()` and `b()` follows `a()`?"*. This explains why method `semi_compliant_1` is recognized as a non-complying method although there is a path through the method that complies with the protocol.

Evaluation Template terms provide a more descriptive means to express control flow characteristics, but they share the same limitation as the control flow traversal predicates they compile to. It is not possible to express the existential path query with a complement *"does there exist a path through `?m` on which anything but `b` is executed between `a()` and `c(?arg)`?"*. We intend to address this limitation in future work (cf. Section 5.5.2).

7.4.4 Expressing Data Flow Characteristics

In this section, we revisit the example-based specifications for methods that comply with the above protocol and potentially enhanceable `for`-statements to express their data flow characteristics.

Enhanceable `for`-Statements Revisited

Expressing the data flow characteristics of enhanceable `for`-statements only requires minor changes to the original template terms depicted in Figure 7.9. It suffices to substitute variable *?iterator* for variable *?hasNextReceiver* (lines 2 and 8) as well as for variable *?nextReceiver* (lines 3 and 9). The data flow characteristics require methods `hasNext()` and `next()` to be invoked on the same iterator. Figure 7.13 depicts the quantified solutions to the adapted queries. The queries have the same solutions as the LMP query in Figure 6.9 that used `equals:/2` to unify both receivers explicitly. Only the truth degrees listed in the second and third column differ.

The second column of Figure 7.13 lists the truth degrees for the solutions to the adapted query with the `jtStatement/1` template term. They correspond to the truth degrees for the solutions to the original query (listed in the second column of Figure 7.9) multiplied by the unification degrees of the `hasNext()` and `next()` receivers (listed in Figure 6.9). According to the domain-specific unification procedure, the bindings for both occurrences of variable *?iterator* are consistent if they are the same AST node or if they are expressions in a may-alias or must-alias relation. The associated unification degrees are 1, 0.9 and 0.5 respectively (cf. Section 6.4.1). The `for`-statement in method `enhanceable_4`, for instance, has a

```
1  if jtClassDeclaration(?class,?interpretation){
2       class ProtocolExample {
3         ?m := !static ?type::jtType ?name(?paramList){
4                 ?a := a();
5                 ?c := c(?arg);
6               }
7       }
8     },
9     absolutelyNot(jtMethodDeclaration(?m,?interpretation){
10      ?modList ?type ?name(?paramList){
11        ?a := a();
12        b();
13        c(?arg);
14      }
15    }),
16    ?modList equals: ?mods,
17    ?m methodDeclarationHasBody: ?block
```

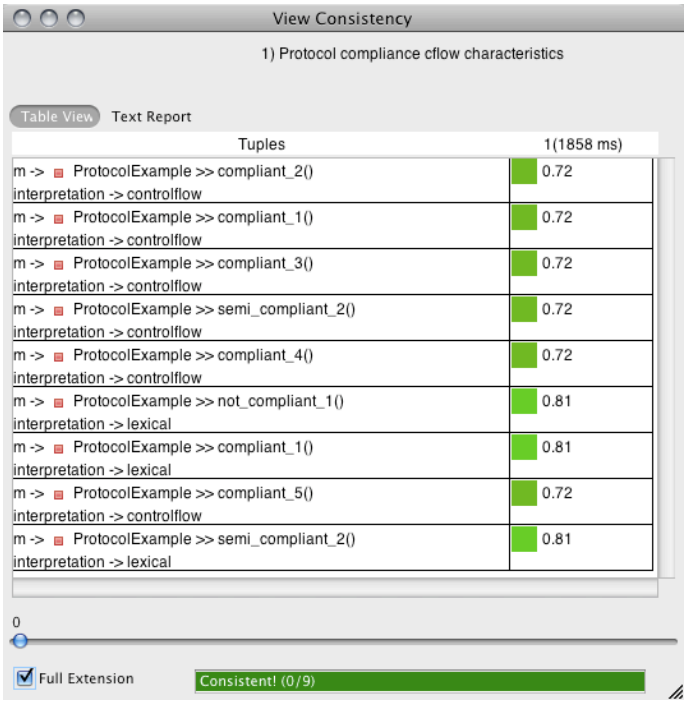


Figure 7.12: Example-based spec. for the control flow char. of complying methods.

maximum associated truth degree of 0.45 = 0.9 × 0.5. Here, 0.9 is the upper bound for solutions identified under the lexical interpretation of the `jtStatement/1` term and 0.5 is the unification degree of receiver expressions `((Iterator) temp)` and `i` which are in a may-alias relation.

The third column lists the truth degrees for the solutions to the adapted query with the `jtMethodDeclaration/1` template term. In contrast to the truth degrees for the solutions to the adapted `jtStatement/1` query, they do not correspond to the truth degrees for the original `jtMethodDeclaration/1` query multiplied by the unification degrees of the *?iterator* occurrences. Conceptually, the template term is resolved using a different fuzzy logic rule under each interpretation. The listed truth degrees correspond to the truth of a conjunction of goals (i.e. the minimum of their truth degrees) multiplied by the truth degree associated with the example-based interpretation under which the template term compiles to these goals. Of these generated goals, the following may influence the truth degree for a solution (i.e. can succeed with a truth degree less than 1):

```
1 ...,
2 ?Var687354 blockIsLexicalCandidateForAmountOfActualStatements: [1],
3 ...,
4 ?Var687358 equals: methodInvocation(?iterator,?,simpleName(['hasNext']),?),
5 ?Var687361 equals: methodInvocation(?iterator,?,simpleName(['next']),?),
6 ...
```

The first goal compares the number of statements in a `MethodDeclaration` instance with the number of statements in the `jtMethodDeclaration/1` term. It uses predicate `isEqualToOrGreaterThanButRelativelyCloseTo:/2` which, for the listed solutions, succeeds with a truth degree of at least 0.9 (cf. Section 7.4.1). Through the occurrences of *?iterator*, the second and third goals assert that the receivers of the `hasNext()` and `next()` invocations are the same. The truth degree of the third goal is the unification degree of both receivers. It unifies the `next()` invocation AST node (bound to generated variable *?Var687361*) with a structurally equivalent compound term to access its receiver child. This entails unifying the receiver child node with the binding for *?iterator* (already established by the second goal). The third goal therefore succeeds with a truth degree of 1 (same AST node), 0.9 (in a must-alias relation) or 0.5 (in a may-alias relation) —depending on the domain-specific unification extension according to which both receivers unify.

Evaluation The above explains why the truth degrees in the third column do not correspond to the original truth degrees multiplied by the unification degrees of the *?iterator* occurrences. In the original query, the first goal succeeded with the smallest truth degree of all generated goals and therefore determined the truth degree for each solution (listed in the third column of Figure 7.9). In the adapted query, the influence of the first goal is lost. The listed truth degrees correspond only to the truth degree of the last goal, multiplied by the truth degree associated with the lexical interpretation (i.e. 0.9). This is because of the way fuzzy SOUL quantifies conjunction (i.e. the minimum truth degree of all goals in the conjunction). The future work for the fuzzy logic cornerstone therefore includes investigating other quantifications (e.g. product) that give rise to a more refined ranking (cf. Section 6.8).

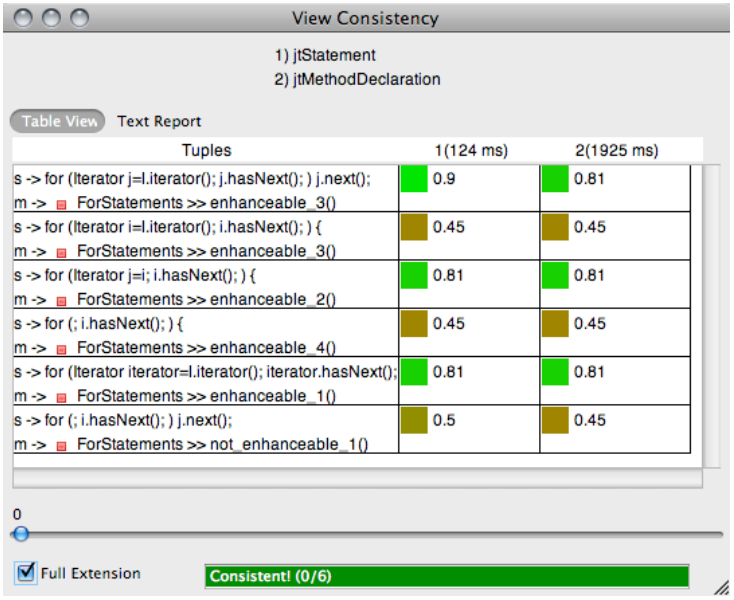


Figure 7.13: Quantified solutions to example-based spec. for enhanceable `fors`.

The Protocol Revisited

The data flow characteristics of the protocol require that the invocation of method `c(Object)` takes the result of a prior invocation of method `a()` as its argument (cf. Section 5.3.3). Figure 7.12 depicts an example-based specification for the control flow characteristics of methods that comply with the protocol. Expressing their data flow characteristics requires only minor changes to this specification. It suffices to substitute variable *?a* for variable *?arg* in the second template term and to substitute either of the following bodies for the body of method *?m* in the first template term:

```
1  { ?a := a(); ?c := c(?a); }

2  { c(a()); }
```

The second and third column of Figure 7.14 list the truth degrees for the solutions to the specification with the first and second modification respectively. Under the control flow interpretation, both specifications identify the same solutions. The first specification explicitly requires method `a()` to be invoked before method `c(Object)` and the argument of the latter invocation to unify with the former invocation. The second specification implies the same requirements because the arguments to an invocation are always evaluated before the invocation itself. We refer to Section 6.6.4 for an in-depth discussion of their solutions as both specifications roughly correspond to the LMP query discussed there.[14]

[14]Note that the truth degrees for the example-based specifications do not correspond to the truth degrees for the LMP query multiplied by the truth degree associated with the control flow interpretation (i.e. 0.8). Method `compliant_2`, for instance, was identified by the LMP query with a truth degree of 1 (cf. the second column in Figure 6.10). The argument to the invocation of `c(Object)` and the invocation of `a()` unify with a unification degree of 1 (i.e. they are the same AST node). The example-based spec-

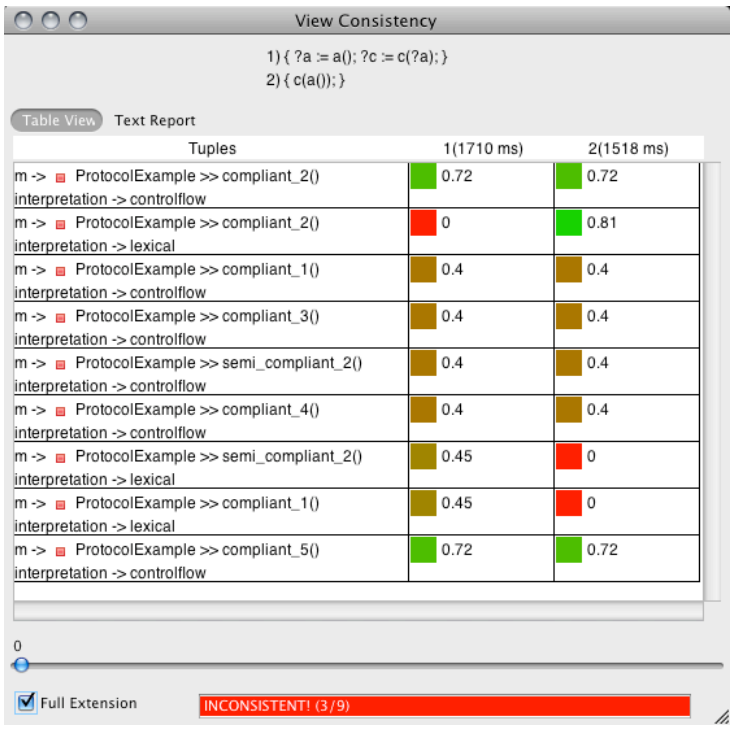


Figure 7.14: Quantified solutions to example-based spec. for complying methods.

Under the lexical interpretation, the solutions to the first and second specification differ. For instance, the second specification does not identify method `compliant_1` (i.e. there is a 0-entry in the third column) while the first specification does. This is because the first specification merely requires both invocations to reside in the same method. The second specification requires the invocation of `a()` to reside lexically within the invocation of `c(Object)`. As a result, method `compliant_2` is the only method identified under the lexical interpretation by the second specification. Clearly, the specifications only exemplify the intended control flow characteristics under the control flow interpretation. An additional condition *`?interpretation`* `equals: controlflow` would exclude the other interpretations from being considered.

Evaluation Expressing the data flow characteristics of the pattern required only minor changes to the example-based specification that expressed its control flow characteristics. However, this is due to the domain-specific unification procedure. The example-based specification is an improvement on the original LMP query in that it exemplifies the pattern's control flow characteristics through a descriptive code excerpt.

ifications, in contrast, identify this method with a truth degree of 0.72. Again, this is because of a goal with predicate `blockIsLexicalCandidateForAmountOfActualStatements:/2` (see the discussion on the example-based specification for enhanceable `for`-statements).

7.5 Open Implementation

The translational semantics of the example-based interpretations are realized by logic rules that implement predicate `underInterpretation:compilesTo:forResult:/4`. These rules comprise the meta-level interface through which additional interpretations can be defined (cf. Section 4.6.4).

A compile-time instance of SOUL invokes this predicate to obtain the goals that should be used to resolve each template term (cf. Figure 4.1). These goals are stored in the object that implements the term. The stored goals are subsequently used to resolve the term at run-time. This way, goals that use a template term do not incur a performance overhead from the compilation step.

Figure 4.14 depicts the rules that implement the translational semantics for `return`-statements in template terms. These rules are publicly available as the Smalltalk package "`Soul-JavaTemplates`" that can be downloaded from the SOUL website [Sou08].[15] They provide a complete and formal account of the standard example-based interpretations introduced in Section 7.2.

The package also defines a library of auxiliary predicates that facilitate defining additional interpretations. The translational semantics of a modifiers lists, for instance, are implemented in a generic manner. Lines 9–16 of Figure 7.6 result from invoking a higher-order list compilation predicate. Its arguments determine how elements in the lists should be compiled (`modifierUnderInterpretation:compilesTo:forResult:/4`), how elements of the matching `ASTNode$NodeList` instance should be quantified over existentially (`collectionContains:andAlso:/3`) and how such goals should be negated (a combination of `absolutelyNot/n` and `contains:/2`). Due to space restrictions, we have to refer the reader to the SOUL website [Sou08] for more information.

7.6 Limitations of the Instantiation

Template terms share the technical limitations of the control flow traversal predicates they compile to under the control flow interpretation. It is not possible to express an existential path query with a complement (cf. Section 5.5.2).

We discuss another technical limitation of template terms below. The concluding chapter discusses the open research questions related to the corresponding example-based specification cornerstone.

The Grammar for Source Code Excerpts is Constructed in an Ad-Hoc Manner

The source code excerpts in template terms are parsed by a Definite Clause Grammar [PW80] (cf. Section A.1). The DCG rules describe the concrete syntax of Java extended with logic variables and a minimum of non-native syntax (cf. Section 7.1.3).

We constructed this grammar by hand —carefully inserting goals such as `jtMetaVariable/1` where logic variables can be used. Their location has a profound impact on the translational semantics of the example-based interpretations. Consider the template term in the following query:

```
1  if jtClassDeclaration(?class) { class ?name { ?member } }
```

[15] The package depends on packages "`Soul-Cava`" and "`Soul-FuzzyLogic`" which roughly correspond to Chapter 5 and Chapter 6 of this dissertation.

Under all but the syntactic interpretation, it quantifies over all class declarations and all of their member declarations. Depending on the location of the `jtMetaVariable/1` goal that recognizes variable *?member*, different ASTs correspond to the excerpt. The DCG rules on lines 2–5 below, construct the single AST depicted on the first line:[16]

```
1 classDeclaration(e,?name,e,e,classBody(<classBodyDeclaration(?member)>))

2 jtClassMemberDeclaration(?var) --> jtMetaVariable(?var)
3 jtClassMemberDeclaration(?decl)--> jtMethodDeclaration(?decl)
4 jtClassMemberDeclaration(?decl) --> jtFieldDeclaration(?decl)
5 ...
```

To ensure that the template term quantifies over all class declarations and their member declarations, the translational semantics of the example-based interpretations includes a disjunction such as "`or(?member isMethodDeclaration, ...)`"

The DCG rules on lines 4–8 below, in contrast, push the `jtMetaVariable/1` goal further down in the grammar. They produce the forest of ASTs depicted on lines 1–3:

```
1 classDeclaration(e,?name,e,e,classBody(<classBodyDeclaration(fieldDeclaration(?member))>))
2 classDeclaration(e,?name,e,e,classBody(<classBodyDeclaration(methodDeclaration(?member))>))
3 ...

4 jtClassMemberDeclaration(?decl) --> jtMethodDeclaration(?decl)
5 jtClassMemberDeclaration(?decl)--> jtFieldDeclaration(?decl)
6 jtMethodDeclaration(methodDeclaration(?var)) --> jtMetaVariable(?var)
7 jtFieldDeclaration(fieldDeclaration(?var)) --> jtMetaVariable(?var)
8 ...
```

We implemented the first option to disambiguate the grammar as much as possible. The disadvantage is that the implementation of the translational semantics is more involved. In future work, we want to investigate a more disciplined and automatic conversion of the grammar for the base programming language to a grammar for template terms.

7.7 Conclusion

In this chapter, we discussed the instantiation of the example-based specification cornerstone. It enables exemplifying a pattern through code excerpts that correspond to the prototypical implementation of its essential characteristics. This obviates the need to explicitly quantify over the reified program representation to express these characteristics.

Concretely, we integrated code excerpts as template terms in the fuzzy version of SOUL. They are specified in the concrete syntax of the base program, augmented with logic variables to indicate points of variation among a pattern's instances.

AST nodes match a template term under a particular example-based interpretation. We defined three standard interpretations: the syntactic, lexical and control flow interpretation. The points of variation among the matches for a template term differ under each interpretation. Under the control flow interpretation, for instance, the control flow characteristics of the source code excerpt exemplify the intended matches. The common design principle of the standard interpretations is

[16] Out of space considerations, symbol `e` substitutes for symbol `epsilon` which denotes the empty string.

that matches have to exhibit all exemplified characteristics, but that what is not exemplified cannot constrain the matches further. This way, they realize the example-based semantics of template terms. The domain-specific unification procedure complements the example-based interpretations. It ensures that occurrences of the same variable are consistent across the terms in a specification. All interpretations therefore allow different implementations of a data flow characteristic in their matches.

We discussed the translational semantics of each interpretation. Conceptually, each example-based interpretation transforms the code excerpt of a term into a fuzzy logic rule. The generated rules are used to resolve the template term, which explains the truth degrees associated with its matches. These cannot exceed 1, $\frac{9}{10}$ and $\frac{8}{10}$ under the syntactic, lexical and control flow interpretation respectively. This ranking reflects the projected similarity of the solutions to the code excerpt of the term. The properties of the solution itself further refine this upper bound. For instance, if it required a domain-specific unification that could introduce false positives.

We defined the translational semantics of the standard interpretations. Moreover, we discussed the meta-level interface through which the translational semantics of additional interpretations can be implemented as logic rules.

We have shown that composing template terms through logic connectives allows for finer-grained control over their matches. We have also presented a minimum of non-native syntax operators without which many example-based specifications would be less concise. Because these detract from a term's resemblance to actual code, however, we have shown how each non-native operator can be eliminated.

Finally, we have shown that this cornerstone overcomes the quantification-related shortcomings of LMP by specifying the patterns that are representative for each kind of pattern characteristic in an example-based manner.

CHAPTER 8

VALIDATION: DETECTING PATTERNS USING EXAMPLE-BASED QUERIES

This chapter validates our example-driven approach to pattern detection by demonstrating that it fulfills the criteria for a general-purpose pattern detection tool that we identified in Section 2.6. We apply our approach, as instantiated in the previous chapters, to several of the software patterns introduced in Section 2.1. As discussed in Section 2.3, detecting these patterns has valuable applications throughout the development process. We demonstrate that they can be specified as descriptive example-based specifications in a uniform language —even though many are heterogeneously characterized which would require developers to use multiple tools with diverse specification languages (cf. Section 3.6.1 for an evaluation of the specification languages of the state of the art). In addition, we present guidelines that can be followed by developers when exemplifying other software patterns such as the application-specific patterns that have served as running examples for the previous chapters. An explicit evaluation of our approach on the criteria for a general-purpose pattern detection tool concludes this chapter.

8.1 Detecting Design Patterns

We detected 7 of the 23 design patterns introduced by Gamma et al. [GHJV94] in an example-driven manner. We selected patterns from each design pattern category: creational patterns (*Singleton*, *Factory Method* and *Prototype*), structural patterns (*Composite* and *Decorator*) and behavioral patterns (*Observer* and *Template Method*). The selected patterns have their instances explicitly documented in JHOTDRAW 5.1 [jHo07] —which allows assessing the pattern detection results.

Section 8.1.1 presents the example-based specifications for the selected design patterns. Section 8.1.2 subsequently uses those specifications to detect design pattern instances in two Java programs. Both programs are publicly available: JHOTDRAW 5.1 [jHo07] and an academic implementation of each design pattern by Hannemann and Kiczales [HK02]. The former program, Gamma's port of the Smalltalk

HOTDRAW framework [Joh92] to Java, enables assessing our pattern detection results. Each pattern instance is documented in its code. Moreover, Riehle's dissertation [Rie00] provides role-model enhanced class diagrams of the JHOTDRAW 5.1 framework.[1] The latter program ensures that our example-based specifications are not specific to the JHOTDRAW implementations. Appendix B.1 details some statistics about the size of both programs.

8.1.1 Example-based Specifications

Figure 8.1 and Figure 8.2 depict the example-based specifications for the following design patterns. The logic variables in each specification are named after the classes that participate in each pattern and their members as described in [GHJV94].

The Singleton design pattern *"ensures that a class has one instance and provides a global point of access to it"*. Figure 8.1 exemplifies three prototypical implementations of the pattern in Java. All implementations have a static field *?uniqueInstance* and a static, parameter-less method *?instance* that returns this field.

The first exemplified implementation has no public constructor. It assigns the field a new instance of *?singleton* in the initializer expression of the corresponding field declaration.

The second implementation assigns the field through an *?initializer* method. Note that we split the specification in two template terms. This allows a single method to match both *?initializer* and *?instance*. For instance, a method that initializes *?uniqueInstance* the first time it is invoked (i.e. lazy initialization). Exemplifying both methods in a single class declaration template would require matching classes to feature at least two distinct methods.

The third exemplified implementation requires *?singleton* to have at least one public constructor. All public constructors are moreover required to assign the current object to *?uniqueInstance*. Note that, under the control flow interpretation, this assignment may reside in another constructor or method that is invoked from the public constructor.

The Template Method design pattern *"defines the skeleton of an algorithm in an operation, deferring some steps to subclasses"*. Its specification is depicted in the top-right corner of Figure 8.1. The specification relies on the domain-specific unification of a method invocation name and the name of the invoked method declaration. Note that the *?concreteClass* participant of the pattern is required to implement both abstract methods called by the template method of the *?abstractClass* participant. This corresponds to the pattern's structure in [GHJV94]. A specification with non-native operator `*{` would allow those implementations to be inherited.

The Observer design pattern *"defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"*. The bottom-right corner of depicts its specification. The subject participant is exemplified as a class with a collection of *?observers* to which an *?observer* can be added through method *?addObserver*.

Note that we have used different variables for the formal parameter names of methods *?removeObserver* and *?addObserver*. Otherwise, the specification

[1] In these diagrams, classes are annotated with the participant roles they play in a design pattern.

would require that at least one observer is added to and removed from a subject at run-time. According to the domain-specific unification procedure, formal parameter names only unify if they are in a may-alias relation.

Method *?notifyObservers* notifies the subject's observers of a state change. Rather than enumerating the different ways in which the *?observers* field can be iterated through, the specification exemplifies the method as one with two successively evaluated instructions. The first evaluates to the *?observers* field and the second invokes a method on an *?observer* that has been added to this field through method *?addObserver*.

The observer participant of the pattern is exemplified as a class in which the invoked method resides. Note that this already constrains *?observerClass* to a class declaration in the sub-type hierarchy of *?observerType*. No additional conditions are therefore required to express this constraint.

The depicted specification consists almost entirely of logic variables. These indicate explicit points of variation which are constrained by the domain-specific unification procedure.

The Decorator design pattern *"attaches additional responsibilities to an object dynamically"*. Its specification is depicted in Figure 8.2. Lines 7–24 exemplify the decorator participant as a class with a method *?operation* and a field of type *?componentType*. This *?decoratorClass* has to reside in the sub-type hierarchy of *?componentType*.[2]

Lines 4–6 identify the pattern's concrete decorator participant through an auxiliary predicate. The predicate is implemented by the fuzzy logic rules on lines 25–45. Their bodies require a subclass of *?decoratorClass* to override method *?operation* such that it invokes a method with the same name on the decorated object *?component*. The first rule requires that the original arguments are passed along. The second rule allows different arguments. Its weight of $\frac{9}{10}$ ensures a lower truth degree for the concrete decorator participants it identifies.

The Prototype design pattern *"specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype"*. The top-right corner of Figure 8.2 exemplifies the pattern's prototypical implementation in Java. The client participant is implemented as a class with a field *?prototypeInstance* on which a `clone` method is invoked. The prototype participant therefore has to implement interface `Cloneable`. Note that the specification links the client to the prototype through the left-hand sides of two non-native `:=` operators: the method invocation and the invoked method.

The Composite design pattern *"lets clients treat individual objects and compositions of objects uniformly"*. A specification for this pattern is depicted in the middle of Figure 8.2. The composite participant is exemplified as class with a field *?children* to which objects of type *?componentType* are added through a method *?add*. Lines 12–13 moreover require *?compositeClass* to reside in the sub-type hierarchy of *?componentType*. They are an LMP alternative to lines 13–24 of the specification for the *Decorator* pattern.

[2] The class can either extend the class (or a sub-class thereof) that declares this type, implement the interface (or a sub-inerface thereof) that declares this type, or extend a class (or a sub-class thereof) that implements an interface (or a sub-interface thereof) that declares this type.

The Factory Method design pattern defines *"an interface for creating an object, but lets subclasses decide which class to instantiate"*. Its specification is depicted in the bottom-right corner of Figure 8.2. The creator participant is exemplified as a class in which method *?operation* invokes the factory method on the current object. Note that the specification does not state where the factory method is declared (e.g. in the creator, an extended super-class or an implemented interface). It should only be possible to invoke the method on an instance of the creator participant. The concrete creator participant can be either the creator participant itself or a sub-class thereof. It should declare (and possibly override) the factory method. This method has to return a new instance of concrete product *?concreteProductClass*. Note that the specification does not exemplify this characteristic through a statement "`return new ?concreteProductClass(?aList);`". That would, under the syntactic and lexical interpretation, require the instance to be instantiated within the expression operand of the statement. Instead, the specification requires the operand to unify with such an instance that is identified through non-native operator `:=`.

Evaluation The above specifications evidence the expressiveness of our specification language. Within fuzzy as well as regular logic rules, they combine template terms with other logic terms to implement user-defined predicates. They contain higher-order logic goals such as `forall/2`, but also invoke Java and Smalltalk methods on variable bindings.

Of course, alternative specifications are possible. The somewhat intricate Smalltalk term in the specification for the *Composite* pattern, for instance, can be replaced by a predicate of the CAVA library.

Design patterns describe proven object-oriented solutions to common design problems. As such, they can be implemented in many different ways. Consider the *Observer* pattern. The subject participant can push information about its entire state towards its observers (i.e. push model). Another implementation has the subject notify its observers, upon which the observers only pull the information that is relevant to them (i.e. pull model). A prototypical Java implementation corresponds to each model. The depicted specification covers the prototypical implementation of either model by only exemplifying their common characteristics. The specification for the *Singleton* pattern, on the other hand, enumerates three prototypical Java implementations of the pattern. The *Decorator* specification moreover ranks the prototypical implementations of the concrete decorator participant it enumerates. *We do not claim that the above specifications cover all prototypical implementations. However, our detection mechanism recognizes different implicit variants (i.e. implied by the semantics of the programming language) of each exemplified prototypical implementation (cf. Section 8.1.2).*

Overall, the above design patterns were straightforward to specify in an example-driven manner. The specifications for the *Template Method*, *Prototype* and *Observer* patterns, for instance, consist exclusively out of template terms with a minimum of non-native syntax. The cardinality constraints of the *Singleton*, however, were less straightforward to specify (i.e. the for-all and at-least-one requirement expressed through lines 41–50). We will encounter similar problems in Section 8.2 when specifying the cardinality constraints of the μ-patterns.

Figure 8.1: Example-based specifications for the *Singleton, Template Method* and *Observer* design patterns.

Figure 8.2: Example-based specifications for *Decorator*, *Prototype*, *Composite* and *Factory Method* design patterns.

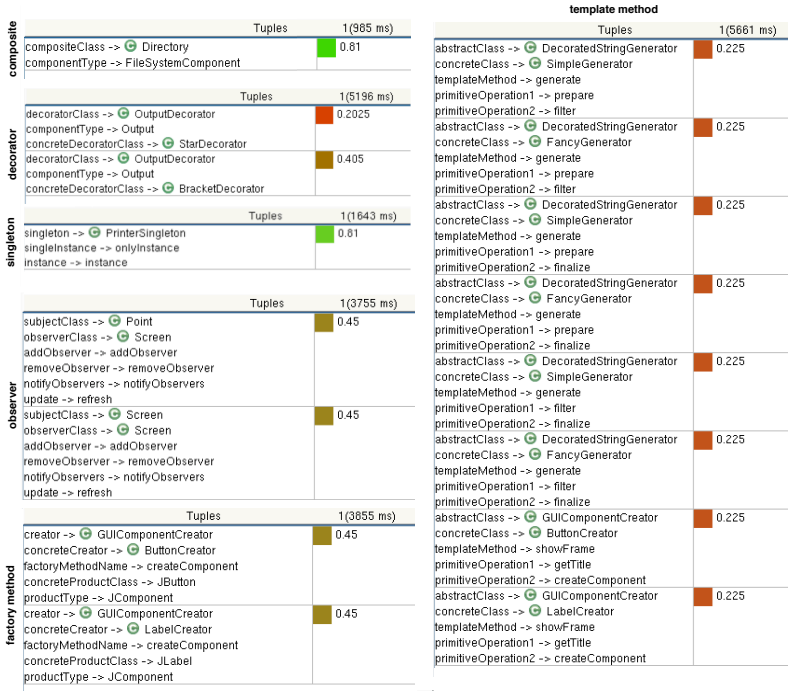


Figure 8.3: Design patterns detected in the academic program [HK02].

8.1.2 Experimental Results

We subsequently used the above specifications to detect design pattern instances in JHOTDRAW 5.1 [jHo07] and in Hannemann's academic program [HK02]. We will discuss the results for these base programs separately.

Except for the *Singleton*, we resolved all pattern specifications under the lexical interpretation. None of the method declarations in these specifications exemplify complex control flow characteristics. Only the specification for the *Template Method* and *Observer* pattern exemplify a method with multiple instructions, but these do not need to be matched inter-procedurally.[3] Inter-procedural matching is only necessary for the third exemplified implementation of the *Singleton* (see above). Not having to consider the inter-procedural control flow interpretation reduces the running time of the experiments significantly.[4]

1/ Results for Hannemann's Program

Figure 8.3 depicts the results for the example-based specifications on Hannemann's academic base program [HK02]. Table 8.1 lists the precision and recall ratios (cf.

[3] In fact, evaluating the specification for the *Template Method* under the control flow interpretation would lead to false positives as a *Template Method* should call its primitive operations directly. In future work, additional non-native syntax could be introduced to specify which instructions in an exemplified method declaration should not be matched inter-procedurally under the control flow interpretation.

[4] To determine whether instructions may be executed consecutively, our prototype performs successive control flow graph traversals. We intend to address this shortcoming by adopting state of the art model checking algorithms for the translational semantics of the control flow interpretation (cf. Section 5.5.2).

	documented	reported	true positives	false positives	missed	precision	recall
Prototype	1	0	0	0	1	NA	0
Template Method	3	5	3	2	0	0.6	1
Composite	1	1	1	0	0	1	1
Decorator	2	2	2	0	0	1	1
Singleton	1	1	1	0	0	1	1
Observer	2	2	2	0	0	1	1
Factory Method	2	2	2	0	0	1	1

Table 8.1: Precision and recall of design patterns in the academic program [HK02].

Section 2.4.3) attained by these specifications.[5] For all but the *Prototype* and *Template Method* design patterns, *the specifications recalled all pattern instances without false positives.*

Missed Instances We recalled all pattern implementations in the academic program, except for the one of the *Prototype* pattern.

- In the academic implementation of the *Prototype*, there is no class that corresponds to the pattern's client participant. Specifically, the implementation does not have a class that clones one of its fields. The instance would have been recognized if line 7 were omitted from the specification in Figure 8.2. However, the current specification is closer to the prototypical implementation described in [GHJV94].

False Positives Of all patterns, only the results for the *Template Method* include a false positive:

- Method `generate` in `DecoratedStringGenerator` is correctly recognized as a *Template Method* that successively invokes primitive operations `prepare`, `filter` and `finalize`. These operations are overridden in `SimpleGenerator` and `FancyGenerator`. Note that our specification exemplified only two primitive operations. In the solutions where the first operation is bound to `prepare`, the second operation is therefore either bound to `filter` or `finalize`. Method `showFrame` in `GuiComponentCreator` belongs to a *Factory Method* implementation, but exhibits all of the specified characteristics of a *Template Method*. It is therefore not a false positive with respect to the specification.

  All reported *Template Method* instances have a low associated truth degree. They required the names of a method declaration and method invocation pair to unify of which the former cannot be invoked by the latter according to the points-to set of the receiver (unification degree of $\frac{1}{2}$), but only according to the static type of the receiver (unification degree of $\frac{1}{4}$). This is already clear from the specification which exemplifies the primitive operations as abstract methods.

Pattern Instances with Different Truth Degrees The instances of the *Decorator* pattern have been identified with different truth degrees:

[5] In the table, each unique tuple of <*?abstractClass*, *?concreteClass*, *?templateMethod*> bindings is considered an instance of the *Template Method*.

- The two *Decorator* instances are identified with different truth degrees. The goals on lines 7–24 identify their "decorator" participant `OutputDecorator` with the same truth degree.[6] The difference is therefore due to different truth degrees for the goal on lines 4–6 which identifies their "concrete decorator" participant: 0.405 for `BracketDecorator` versus 0.2025 for `StarDecorator`. For both, the goal is resolved using the fuzzy rule on lines 35–45 of the specification. To understand the difference in truth degrees, we have to consider that a `StarDecorator` instance is configured to decorate a `BracketDecorater`:

```
1 Output original = new ConcreteOutput();
2 Output bracketed= new BracketDecorator(original);
3 Output stared   = new StarDecorator(bracketed);
4 stared.print("<String>");
```

  Method `BracketDecorator»print`[7] is called at run-time from within method `StarDecorator»print`. Method `StarDecorator»print`, in contrast, is not called from within any `print` method. The points-to set for *?component* on line 41 therefore does not include a `StarDecorator` instance in solutions where *?concreteDecoratorClass* is bound to `StarDecorator`. As a result, the bindings for *?operation* on line 40 and 41 unify according to the static type of *?component* within `StarDecorator»print`. Within `BracketDecorator»print`, they unify according to the points-to set of *?component*. This explains the different truth degrees for both "concrete decorator" participants: $0.405 = 0.9 \times 0.9 \times 0.5$ versus $0.2025 = 0.9 \times 0.9 \times 0.25$. The first factor of each product stems from the fuzzy rule. The second factor corresponds to the lexical interpretation under which the template term in the rule is resolved. The third factor is the degree to which the *?operation* bindings unify.

Interesting Implementation Variants The following highlights some interesting implementation variants that have been detected:

- The *Singleton* implementation is identified by the logic rule on lines 13–31 of the specification. The method through which the unique instance can be accessed, `instance()` lazily initializes the instance. It is therefore bound to *?instance* as well as *?initializer* in the solution:

```
1 public static PrinterSingleton instance() {
2   if(onlyInstance == null) {
3     onlyInstance = new PrinterSingleton();
4   }
5     return onlyInstance;
6 }
```

  Note that we would not have identified this implementation variant if we had not exemplified the *?instance* and *?initializer* methods in different class declaration template terms.

- The *Composite* implementation is recalled without false positives. Only class `Directory` is a composite for the `FileSystemComponent` type (an interface). Note that the field access in method *?add* of the specification is exemplified with the current object as the implicit base expression, while method `add` has the current object as the explicit base for the field access:

[6] Specifically, a truth degree of $0.45 = 0.9 \times 0.5$ where 0.9 is the truth degree for the lexical interpretation and 0.5 is the unification degree for the *?component* occurrences.

[7] We are using the traditional Smalltalk notation to refer to a method in a class.

	documented	reported	true positives	false positives	missed	precision	recall
Prototype	3	7	3	4	0	0.43	1
Template Method	3	5	1	4	2	0.2	0.34
Composite	1	1	1	0	0	1	1
Decorator	2	1	1	0	1	1	0.5
Singleton	2	2	2	0	0	1	1
Observer	3	NA	NA	NA	3	NA	NA
Factory Method	4	NA	NA	NA	4	NA	NA

Table 8.2: Precision and recall of design patterns in JHOTDRAW 5.1 [jHo07].

```
1 public void add(FileSystemComponent component) {
2   this.children.add(component);
3 }
```

- The solutions for the *Observer* pattern include class `Screen` twice: once with `Point` as a subject and once with itself as a subject. Neither solution is a false positive. Particular instances of `Screen` are configured to observe other `Screen` instances. The `Screen` class implements both a `ChangeSubject` and a `ChangeObserver` interface. The corresponding solution is therefore not a false positive.

  Note that these *Observer* instances have only been identified because of the domain-specific unification procedure. Within method `notifyObservers`, the receiver of message `refresh` (bound to the second occurrence of *?observer*) is in a may-alias relation with the parameter of method `addObserver` (bound to the first occurrence of *?observer*):

```
1 public void notifyObservers() {
2   for (Iterator e = observers.iterator() ; e.hasNext() ;) {
3       ((ChangeObserver)e.next()).refresh(this);
4   }
5 }
```

Remaining Results The following results have not yet been discussed:

- Classes `LabelCreator` and `GUIComponentCreator` have been identified correctly as the only concrete creator participants of a *Factory Method*. Both override method `createComponent`.

2/ Results for JHOTDRAW 5.1

Using the same specifications, we were able to *recall most design pattern instances* in JHOTDRAW 5.1 with *few false positives*. Table 8.2 lists the precision and recall ratios (cf. Section 2.4.3) attained by these specifications.[8] At the end of this section, we discuss how improving these results requires only minor changes to the example-based pattern specifications.

Figure 8.4 and Figure 8.5 depict the design pattern instances as detected in JHOTDRAW 5.1 [jHo07] under the lexical interpretation:

[8] The listed amount of false positives is with respect to the documented pattern instances. They include solutions that can be considered undocumented design pattern instances.

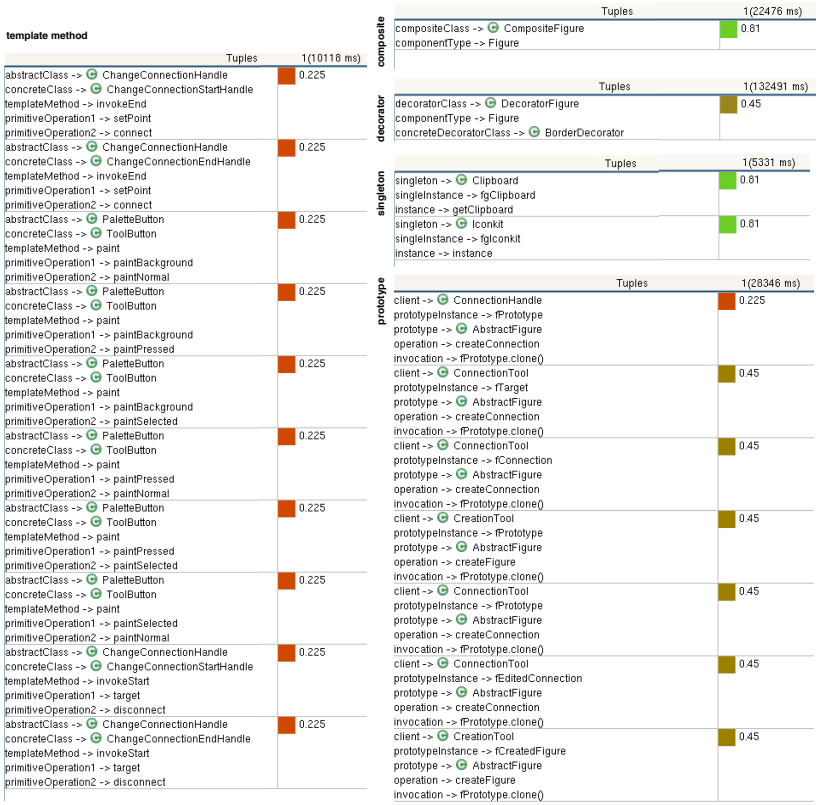


Figure 8.4: Design patterns detected in JHOTDRAW 5.1 [jHo07] (1).

Missed Instances Except for the following, we recalled most design pattern instances in JHOTDRAW:

- The reported class `DecoratorFigure` is the only documented *?decorator* participant of the *Decorator* pattern. However, only its `BorderDecorator` subclass is recognized as a "ConcreteDecorator" participant under the lexical interpretatinon. The methods in its other subclass, `AnimationDecorator`, decorate *?component* indirectly through a super invocation:

```
1 public synchronized Rectangle displayBox() {
2   return super.displayBox();
3 }
```

  Class `AnimationDecorator` is therefore only recognized as a participant under the time-consuming control flow interpretation.

- There are three documented instances of the *Template Method*: `AbstractFigure»moveBy`, `AbstractFigure»displayBox` and `AttributeFigure»draw`. The first two are not included in the solutions because they invoke only one primitive operation (i.e. `basicMoveBy` and `basicDisplayBox` respectively) rather than the required two. Method `AbstractFigure»displayBox` does call two primitive operations (i.e. `drawBackground` and `drawFrame`), but these are declared with an empty

body rather than abstract as required. To cover these instances, our specification should be relaxed.

Although not documented in the code, the depicted solutions are no false positives with respect to the specification and can all be considered instances of *Template Method* to some extent. Method `PalleteButton»paint`, for instance, is correctly recognized as invoking two abstract methods —even under the control flow interpretation:

```
1  public abstract void paintBackground(Graphics g);
2  public abstract void paintNormal(Graphics g);
3  public abstract void paintPressed(Graphics g);
4  public abstract void paintSelected(Graphics g);
5  public void paint(Graphics g) {
6    paintBackground(g);
7    switch (fState) {
8      case PRESSED:
9        paintPressed(g);
10       break;
11     case SELECTED:
12       paintSelected(g);
13       break;
14     case NORMAL:
15     default:
16       paintNormal(g);
17       break;
18   }
19 }
```

- Figure 8.5 depicts the results for the *Factory Method* in a compact manner. Each solution is depicted as a logic list of bindings for the variables in the specification.

  None of the documented *Factory Method* instances is included in the solutions. This is because our specification strictly corresponds to the prototypical implementation of the pattern as described in [GHJV94]. The specification requires an *?operation* within the *?creator* participant to invoke the factory method of the *?concreteCreator* participant —which is not the case for the `jHotDraw` implementations.

  However, all identified methods of which the name is prefixed by "`create`" can be considered undocumented *Factory Method* instances. The other solutions match the specification, but are false positives. Method `PolyLineFigure»displayBox`, for instance, creates a `Rectangle` and is called from within an *?operation* in a super-class, but is not a *Factory Method*:

```
1  public Rectangle displayBox() {
2    Enumeration k = points();
3    Rectangle r = new Rectangle((Point) k.nextElement());
4    while (k.hasMoreElements())
5      r.add((Point) k.nextElement());
6    return r;
7  }
```

False Positives Except for the *Observer* pattern, there are few false positives among the pattern detection results:

- Figure 8.5 depicts an extract of the results for the *Observer* in a compact manner. Each solution is depicted as a logic list of bindings for the vari-

ables in the specification. The complete solutions for all specifications can be downloaded from the SOUL website [Sou08].

The instances of this pattern are documented at the interface level (e.g. interface `DrawingView` observes interface `Drawing`), while our specification identifies the classes that implement these interfaces (e.g. `StandardDrawingView` and `StandardDrawing`). Furthermore, `Figure` sub-types `DecoratorFigure`, `ConnectionFigure` and `CompositeFigure` also observe `Figure` itself. *Because of the large amount of solutions this gives rise to (146), we were unable to inspect each solution in detail.*

A cursory exploration reveals that there are many missed instances. This is because our specification does not cover sophisticated implementations of the push model. The one on `AbstractFigure`, for instance, is not recognized because it involves static calls to an intermediary `FigureChangeEventMulticaster`:

```
1  public void addFigureChangeListener(FigureChangeListener l) {
2    fListener = FigureChangeEventMulticaster.add(fListener, l);
3  }
4  public void changed() {
5    invalidate();
6    if (fListener != null)
7      fListener.figureChanged(new FigureChangeEvent(this));
8  }
```

The less sophisticated implementation on `StandardDrawing`, on the other hand, is recognized successfully. Note how the bindings for the different *?observer* occurrences (i.e. `listener` and `l`) are in a may-alias relation:

```
1  public void addDrawingChangeListener(DrawingChangeListener listener) {
2    fListeners.addElement(listener);
3  }

4  public void figureRequestUpdate(FigureChangeEvent e) {
5    if (fListeners != null) {
6      for (int i = 0; i < fListeners.size(); i++) {
7        DrawingChangeListener l = (DrawingChangeListener)fListeners.elementAt(i);
8        l.drawingRequestUpdate(new DrawingChangeEvent(this, null));
9      }
10   }
11 }
```

There are also many false positives among the solutions. This is mostly because of the heuristics used by the specification to identify methods *?addObserver* and *?removeObserver* (i.e. the regular expressions on lines 19 and 20) and method *?notifyObservers* (i.e. not specifying the way the observers are iterated through). The following method, for instance, matches the specification for *?addObserver*:

```
1   public synchronized void bringToFront(Figure figure) {
2     if (fFigures.contains(figure)) {
3       fFigures.removeElement(figure);
4       fFigures.addElement(figure);
5       figure.changed();
6     }
7   }
```

- Classes `CreationTool`, `ConnectionTool` and `ConnectionHandle` are correctly identified as *?client* participants in the *Prototype* design pattern.

Although the *?invocation* that clones the prototype is correct in each solution, there are some false positives with respect to the *?prototypeInstance* that is being cloned. This is the case for the solutions that involve `ConnectionTool` and the solutions that involve `CreationTool` where *?prototypeInstance* is not the receiver of *?invocation*, but is in a may-alias relation with it.

In `CreationTool`, for instance, `fCreatedFigure` keeps track of a fresh clone of `fPrototype` across the `mouseDown`, `mouseDrag` and `mouseUp` events:

```
1  public void mouseDown(MouseEvent e, int x, int y) {
2    fAnchorPoint = new Point(x,y);
3    fCreatedFigure = createFigure();
4    fCreatedFigure.displayBox(fAnchorPoint, fAnchorPoint);
5    view().add(fCreatedFigure);
6  }
7  public void mouseDrag(MouseEvent e, int x, int y) {
8    fCreatedFigure.displayBox(fAnchorPoint, new Point(x,y));
9  }
10 protected Figure createFigure() {
11   if (fPrototype == null)
12     throw new HJDError("No protoype defined");
13   return (Figure) fPrototype.clone();
14 }
```

Pattern Instances with Different Truth Degrees The results for the *Prototype* have different associated truth degrees:

- Of the reported *Prototype* instances, the one in which `ConnectionHandle` is the client participant has the lowest associated truth degree. This is because the whole-program analyses require an entry point for the analyzed program. No information is available for methods and classes that are not reachable from this point. The JHOTDRAW framework illustrates its API through several sample applications. Class `Connectionhandle` is not referenced from the `main` method of the largest sample application, `JavaDrawApp`, which we have chosen as the entry point for the program analyses. From the perspective of `JavaDrawApp`, the low truth degree is therefore justified.

  Unfortunately, the actual false positives for this pattern (see above) do not stand out among the reported instances. This is because our prototype combines truth degrees and resolution degrees prematurely (cf. Section 6.8). In each solution, the unification degree of the *?invocation* occurrences (ranging from $\frac{1}{4}$ to $\frac{1}{2}$) is lower than the unification degree of the *?prototypeInstance* occurrences (ranging from $\frac{1}{2}$ for the false positives to $\frac{9}{10}$ for the actual instances). The former depends on whether the occurrences unify according to the static or the dynamic type of the receiver, while the latter depends on whether the receiver is an expression aliasing the field or is a field access of this field. As these unifications take place in a conjunction of goals (quantified by minimum), the former unification degrees subsume the latter. Section 6.8 discusses how this limitation of our prototype can be addressed in future work.

Remaining Results The following results have not yet been discussed:

- Class `CompositeFigure` is correctly recognized as a *Composite* for com-

ponents of type `Figure`. The latter is an interface, implemented by `AbstractFigure` which is extended by `CompositeFigure`. There are no other documented instances of this design pattern.

- Classes `ClipBoard` and `IconKit` are correctly identified as instances of the *Singleton* design pattern. They are identified by the first and third rule in the specification respectively. There are no other documented instances of this pattern.

Recalling the JHOTDRAW instances that are missing would require minor changes to the pattern specifications. For instance, by using the control flow interpretation to resolve them (e.g. *Decorator*), by exemplifying additional prototypical implementations (e.g. the *Observer*) or by adhering less strictly to the pattern's description in [GHJV94] (e.g. *Template Method Factory Method*). Moreover, many of the reported false positives could be eliminated by taking the intent of the pattern into account. If the *Factory Method* is applied correctly, for instance, its concrete product should not be instantiated outside of the factory method.

8.2 Detecting μ-Patterns

We specified all 27 μ-patterns introduced by Gil and Maman [GM05]. These patterns primarily describe type declarations of which the members are in straightforward structural relations (cf. Section 2.1). The *Implementor* μ-pattern, for instance, describes a class that exclusively implements abstract methods. Section 8.2.1 discusses the resulting example-based specifications.

We subsequently compared the μ-pattern instances identified by JTL and SOUL on two Java programs. In the first program, we hand-coded instances of each μ-pattern. The second program was the 2008/02/01 implementation of the interpreter for the AMBIENTTALK [Amb]. Appendix B.1 details some statistics about the size of these programs. Section 8.2.2 discusses the pattern detection results.

8.2.1 Example-based Specifications

Table B.2 describes the μ-patterns in more detail. Its entries differ from the natural language descriptions given in Gil and Maman [GM05]. Rather than re-interpreting those descriptions, we translated Cohen, Gil and Maman [CGM06a]'s more precise specifications from JTL [CGM06b] to SOUL. Section 3.5.1 discussed this DataLog-variant with a Java-like syntax. The resulting example-based specifications are listed in Section B.2.

We will only discuss the μ-patterns that highlight some interesting differences between both sets of specifications. Figure 8.6 and Figure 8.7 depict the JTL and example-based specifications for the following patterns:

An Outline is an abstract class of which a declared method (different from `main`) invokes an abstract method of the same class (declared or inherited). The example-based specification for this μ-pattern is depicted at the top of Figure 8.6. The first template term exemplifies a class with a method that invokes *?invokedMethod* on `this`. The second template term exemplifies an abstract method named *?invokedMethod* within this class. Because of the non-native `*{` operator (cf. Section 7.1.1), the template also matches methods that are declared in a super class. The bindings for each occurrence of *?invokedMethod*

factory method (compact results)

<?concreteCreator, ?creator, ?concreteProductClass, ?factoryMethodName,?productType>

Tuples	1(285788 ms)
s -> <[StandardDrawingView],[StandardDrawingView],[selectionZOrdered],[Vector],[Vector]>	0.45
s -> <[PaletteLayout],[PaletteLayout],[minimumLayoutSize],[Dimension],[Dimension]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createColorMenu],[Menu],[CommandMenu]>	0.45
s -> <[DrawApplication],[DrawApplication],[createFileMenu],[Menu],[Menu]>	0.45
s -> <[DrawApplication],[DrawApplication],[createFontStyleMenu],[Menu],[CommandMenu]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[PolyLineFigure],[PolyLineFigure],[displayBox],[Rectangle],[Rectangle]>	0.45
s -> <[AbstractFigure],[PolyLineFigure],[displayBox],[Rectangle],[Rectangle]>	0.45
s -> <[DrawApplication],[DrawApplication],[createToolPalette],[Panel],[Panel]>	0.45
s -> <[AbstractFigure],[PolyLineFigure],[displayBox],[Rectangle],[Rectangle]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createStatusLine],[TextField],[TextField]>	0.45
s -> <[DrawApplication],[DrawApplication],[createColorMenu],[Menu],[CommandMenu]>	0.45
s -> <[DrawApplication],[DrawApplication],[createColorMenu],[Menu],[CommandMenu]>	0.45
s -> <[DrawApplication],[DrawApplication],[createArrowMenu],[Menu],[CommandMenu]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createAlignmentMenu],[Menu],[CommandMenu]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createAttributesMenu],[Menu],[Menu]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createEditMenu],[Menu],[CommandMenu]>	0.45
s -> <[JavaDrawApp],[JavaDrawApp],[createImagesMenu],[Menu],[CommandMenu]>	0.45
s -> <[AbstractFigure],[PolyLineFigure],[displayBox],[Rectangle],[Rectangle]>	0.45
s -> <[JavaDrawApp],[JavaDrawApp],[createAnimationMenu],[Menu],[Menu]>	0.45
s -> <[AbstractFigure],[PolyLineFigure],[displayBox],[Rectangle],[Rectangle]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[AbstractFigure],[PolyLineFigure],[displayBox],[Rectangle],[Rectangle]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createFontMenu],[Menu],[CommandMenu]>	0.45
s -> <[DrawApplication],[DrawApplication],[createFontSizeMenu],[Menu],[CommandMenu]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createContents],[Component],[ScrollPane]>	0.45
s -> <[ShortestDistanceConnector],[ShortestDistanceConnector],[findPoint],[Point],[Point]>	0.45
s -> <[DrawApplication],[DrawApplication],[createDebugMenu],[Menu],[Menu]>	0.45
s -> <[ArrowTip],[ArrowTip],[outline],[Polygon],[Polygon]>	0.225
s -> <[JavaDrawApp],[JavaDrawApp],[createWindowMenu],[Menu],[Menu]>	0.45

observer (extract from compact results)

<?subjectClass, ?observerClass, ?addObserver, ?removeObserver, ?notifyObserver, ?update>

Tuples	1(320552 ms)
s -> <[CompositeFigure],[AbstractFigure],[bringToFront],[remove],[sendToBack],[changed]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[remove],[bringToFront],[changed]>	0.45
s -> <[StandardDrawingView],[StandardDrawing],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[TextFigure],[add],[remove],[replace],[changed]>	0.45
s -> <[StandardDrawingView],[StandardDrawing],[add],[remove],[setDrawing],[removeDrawingChangeListener]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[bringToFront],[sendToBack],[changed]>	0.45
s -> <[StandardDrawingView],[GroupFigure],[add],[remove],[checkMinimumSize],[displayBox]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[bringToFront],[sendToBack],[changed]>	0.45
s -> <[StandardDrawingView],[BorderDecorator],[add],[remove],[checkMinimumSize],[displayBox]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[sendToBack],[bringToFront],[changed]>	0.45
s -> <[StandardDrawingView],[AnimationDecorator],[add],[remove],[checkMinimumSize],[displayBox]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[orphan],[bringToFront],[changed]>	0.45
s -> <[StandardDrawingView],[AbstractHandle],[add],[remove],[findHandle],[containsPoint]>	0.45
s -> <[StandardDrawingView],[RectangleFigure],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[TextFigure],[bringToFront],[orphan],[sendToBack],[changed]>	0.45
s -> <[StandardDrawingView],[TextFigure],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[bringToFront],[orphan],[replace],[changed]>	0.45
s -> <[CompositeFigure],[TextFigure],[add],[remove],[bringToFront],[changed]>	0.45
s -> <[StandardDrawingView],[ElbowConnection],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[bringToFront],[orphan],[sendToBack],[changed]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[bringToFront],[orphan],[add],[addToContainer]>	0.45
s -> <[CompositeFigure],[TextFigure],[bringToFront],[remove],[sendToBack],[changed]>	0.45
s -> <[StandardDrawingView],[PolyLineFigure],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[TextFigure],[add],[remove],[sendToBack],[changed]>	0.45
s -> <[CompositeFigure],[TextFigure],[bringToFront],[remove],[replace],[changed]>	0.45
s -> <[StandardDrawingView],[StandardDrawingView],[addToSelection],[removeFromSelection],[checkDamage],[repairDam	0.45
s -> <[StandardDrawingView],[AbstractFigure],[add],[remove],[addToSelection],[invalidate]>	0.45
s -> <[StandardDrawingView],[StandardDrawing],[add],[remove],[setDrawing],[addDrawingChangeListener]>	0.45
s -> <[CompositeFigure],[TextFigure],[add],[bringToFront],[sendToBack],[changed]>	0.45
s -> <[StandardDrawing],[StandardDrawingView],[addDrawingChangeListener],[removeDrawingChangeListener],[figureRec	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[bringToFront],[replace],[changed]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[sendToBack],[replace],[addToContainer]>	0.45
s -> <[StandardDrawingView],[PolyLineFigure],[add],[remove],[checkMinimumSize],[displayBox]>	0.45
s -> <[CompositeFigure],[TextFigure],[add],[orphan],[bringToFront],[changed]>	0.45
s -> <[StandardDrawingView],[TextFigure],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[TextFigure],[bringToFront],[sendToBack],[replace],[changed]>	0.45
s -> <[StandardDrawingView],[ImageFigure],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[bringToFront],[sendToBack],[add],[addToContainer]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[orphan],[sendToBack],[changed]>	0.45
s -> <[StandardDrawingView],[PolyLineFigure],[add],[remove],[selectionHandles],[handles]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[bringToFront],[orphan],[replace],[addToContainer]>	0.45
s -> <[CompositeFigure],[AbstractFigure],[add],[remove],[bringToFront],[changed]>	0.45
s -> <[StandardDrawingView],[AbstractFigure],[add],[remove],[selectionHandles],[handles]>	0.225
s -> <[CompositeFigure],[TextFigure],[bringToFront],[remove],[sendToBack],[change	

Figure 8.5: Design patterns detected in JHOTDRAW 5.1 [jHo07] (2).

have to unify according to the domain-specific unification procedure. It unifies the name of a method invocation and the name of a method declaration if the former may invoke the latter (cf. Section 6.4.1).

The corresponding JTL specification consists of DataLog goals such as `calls`, `extends` and `declared_by` which quantify explicitly over all entities that exhibit the characteristics exemplified by the template terms. The specification does not resemble a source code excerpt. For instance, there is no "concrete" modifier in Java. The template terms, in contrast, exemplify the difference between a concrete and abstract method.

A Function Pointer is a concrete class without a field that has a single public instance method (all declared or inherited, but not from `Object`). The specifications for this μ-pattern are depicted in the middle of Figure 8.6. The JTL specification uses set quantifier `one` (cf. Section 3.5.1) to express that there should only be a single public instance method. The example-based specification expresses this cardinality requirement through an idiom:

```
1  ?method := public !static !abstract ?t2::jtType ?n2(?p2List) ?s2List
2  ![?method :~= public !static !abstract ?t3::jtType ?n3(?p3List) ?s3List]
```

The first line uses non-native operator `:=` to bind *?method* to a public instance method. The second line uses iterator `:~=` to ensure that there are no public instance methods that do not unify with *?method*. This idiom is valid because complemented template elements (i.e. those preceded by complement operator `!`) are matched after all other template elements have been matched.

The original JTL specification did not feature `!synthetic` goals. We added these such that *Function Pointer* instances can have synthetic members which are present in the bytecode, but not in the source code.[9] Similar goals had to be added to all JTL specifications. In fulfillment of criterion **CDM1**, our approach only reports elements from the source code of the base program. This is also why the example-based specification does not have to exclude members inherited from `java.lang.Object`. The base program does not include the source code for library classes.[10]

A Stateless is a class of which all declared and inherited fields are static and final. The specifications for this μ-pattern are depicted at the bottom of Figure 8.6. The JTL specification uses the set quantifier `->` (cf. Section 3.5.1) to express that each field has to be static and final. The example-based specification expresses that there should be no field that is static and not final, nor a field that is not static and final, nor a field that is not static and not final. All these possibilities had to be enumerated because the `!` operator does not yet support grouping several modifiers (e.g. `![final static]`).

An Immutable is a class of which all declared and inherited instance fields are private (and that has at least one such field). Moreover, the class should have no declared method that assigns any of its declared or inherited fields (i.e. a mutator method). The specifications this μ-pattern are depicted at the top of Figure 8.7. The JTL specification refers to a `put_field` operation in the bytecode of a mutator. The example-based specification exemplifies mutator methods through a familiar source code excerpt.

[9] Synthetic members are generated by the Java compiler. The synthetic field `this$0` of an inner class, for instance, refers to its innermost enclosing instance.

[10] If it did, we could introduce additional non-native syntax similar to the existing `*{` operator.

A Designator is an abstract class or interface without methods or fields (all declared or inherited, but not from `Object`). The specifications for this μ-pattern are depicted in the middle of Figure 8.7. The example-based specification is about twice the size of the JTL specification. This is because it has to exemplify a *Designator* class declaration as well as a *Designator* interface declaration. The JTL specification quantifies over all type declarations through a goal `type`. Apart from this difference, both specifications are similar. Note that interfaces can extend multiple super-interfaces. The example-based specification for *Designator* interfaces therefore uses the `forall/2` predicate to ensure that all are empty.

A Compound Box is a class with a single non-primitive instance field and at least one primitive field (all declared or inherited). The specifications for this μ-pattern are depicted at the bottom of Figure 8.7. The JTL specification uses set quantifier `one` to express the cardinality requirement on the non-primitive instance field. The example-based specification uses a variant of the aforementioned idiom. Java field declarations can declare multiple fields at once. Backtracking over the field declaration template "`?modList ?type ?field = ?init;`" successively binds *?field* to each field declared by the matching field declaration. The example-based specification therefore verifies whether the Compound Box has no other non-primitive field than the one identified by the first template term.

Evaluation Section B.2 lists the example-based specifications for the 21 remaining μ-patterns. Most patterns were straightforward to specify.

Compared to the JTL specifications, *the example-based specifications are closer to actual source code excerpts.* The example-based specifications, for instance, exemplify the difference between a concrete and abstract method (e.g. *Outline*).

Only *cardinality constraints were difficult to express* (cf. *Function Pointer* and *Compound Box* above). These would have been easier to express in a pure LMP specification. Specifying a pattern as a LMP specification, however, requires familiarity with the LMP predicates that reify its characteristics. Although sometimes more verbose (e.g. *Designator*), example-based specifications exemplify these characteristics through familiar source code excerpts.

8.2.2 Experimental Results

We compared the μ-pattern instances identified by JTL and SOUL on a program with hand-coded instances of each μ-pattern and on the 2008/02/01 implementation of the AMBIENTTALK [Amb] interpreter. For each μ-pattern, Table B.1 compares the number of instances identified by SOUL and JTL.

Overall, both sets of specifications identified the same instances. This confirms that the example-based specifications (in SOUL) are equivalent to the LMP specifications (in JTL). However, as discussed above, the example-based specifications are closer to actual source code excerpts (except for the patterns that require cardinality constraints). The complete solutions for all specifications can be downloaded from the SOUL website [Sou08].

Most differences were due to the fact that *JTL analyzes the bytecode of the program and the libraries it relies on, while our approach only analyzes the source*

Outline

```
?class isOutlineUnderInterpretation: ?interp if
  jtClassDeclaration(?class,?interp){
    abstract class ?className {
      ?method := [?modList ?returnType::jtType ?methodName(?paramList){
                    this.?invokedMethod(?argList);
                  }]
    }
  },
  absolutelyNot(?method isMainMethodDeclaration),
  jtClassDeclaration(?class,?interp){
    abstract class ?className *{
      abstract ?invokedType ?invokedMethod(?invokedParamList);
    }
  }
```

```
outline := abstract class is C {
        let main_method := public static void 'main'(_);
        let candidate := concrete method !main_method;
        let declared_by T := T declares #;
        candidate calls M, M abstract & declared_by T;
      },
[C extends T | C is T];
```

Function Pointer

```
?class isFunctionPointerUnderInterpretation: ?interp if
  jtClassDeclaration(?class,?interp){
    !abstract class ?className *{
      ![?modList ?t ?field = ?init;]
      ?method := public !static !abstract ?t2::jtType ?n2(?p2List) ?s2List
      ![?method :~= public !static !abstract ?t3::jtType ?n3(?p3List) ?s3List]
    }
  }
```

```
is_common := X is /java.lang.Object, X declares #;
not_common := ! is_common;
uncommons X := offers X, X not_common;
fptr := !abstract class uncommons: {
          no !synthetic field;
          one public instance !synthetic method;
};
```

Stateless

```
?class isStatelessUnderInterpretation: ?interp if
  jtClassDeclaration(?class,?interp){
    class ?className *{
      ![!static !final ?t ?field = ?init;]
      ![static !final ?t1 ?field1 = ?init1;]
      ![!static final ?t2 ?field2 = ?init2;]
    }
  }
```

```
stateless := class offers: {
        !synthetic field -> static final;
};
```

Figure 8.6: SOUL (left) and JTL (right) specifications for select μ-patterns (1).

```
?class isImmutableUnderInterpretation: ?interp if
 jtClassDeclaration(?class,?interp){
   class ?className *{
     !static ?fieldType ?field = ?initializer;
     ![!private !static ?t ?f = ?i;]
   }
 },
 absolutelyNot(
  jtClassDeclaration(?class,?interp){
    class ?className *{
      ?modList ?anyFieldType ?anyField = ?anyFieldInitializer;
    }
  },
  jtClassDeclaration(?class,?interp){
    class ?className {
      !static ?methodType::jtType ?methodName(?paramList) {
        ?anyField = ?assignedValue;
      }
    }
  })
```

```
mutator:= method { put_field[F,_]; } C declares # & offers F;
inspector := method { get_field[F,_]; } C declares # & offers F;
immutable := offers: {
               !synthetic instance field;
               no !synthetic !private instance field;
             no !synthetic !static mutator;
};
```

```
?class isDesignatorUnderInterpretation: ?interp if
  jtClassDeclaration(?class,?interp){
    abstract class ?className extends ?super {
      ![?modList ?type ?m(?paramList) ?statementList]
      ![?modList ?type ?field = ?init;]
    }
  },
  or([?super isNil],?super isDesignatorUnderInterpretation: ?interp)

?interface isDesignatorUnderInterpretation: ?interp if
  jtInterfaceDeclaration(?interface,?interp){
    interface ?interfaceName extends ?superList {
      ![?modList ?type ?m(?paramList);]
      ![?modList ?type ?field = ?init;]
    }
  },
  forall(?superList contains: ?super,
         or([?super isNil],
             ?super isDesignatorUnderInterpretation: ?interp))
```

```
designator := abstract type uncommons: {
                no !synthetic method;
              no !synthetic field;
};
```

```
?class isCompoundBoxUnderInterpretation: ?interp if
  jtClassDeclaration(?class,?interp){
    class ?className *{
      !static ?type ?field = ?init;
      !static ?primType ?primField = ?primInit;
    }
  },
  ?primType isPrimitiveType,
  absolutelyNot(?type isPrimitiveType),
  absolutelyNot(jtClassDeclaration(?class,?interp){
                  class ?className *{
                    !static ?otherType ?otherField = ?otherInit;
                  }
                },
                [?otherType isPrimitiveType not],
                [?otherField ~= ?field])
```

```
compound_box := offers: {
                  one !primitive !synthetic instance field;
                !synthetic primitive instance field;
};
```

Figure 8.7: SOUL (left) and JTL (right) specifications for select μ-patterns (2).

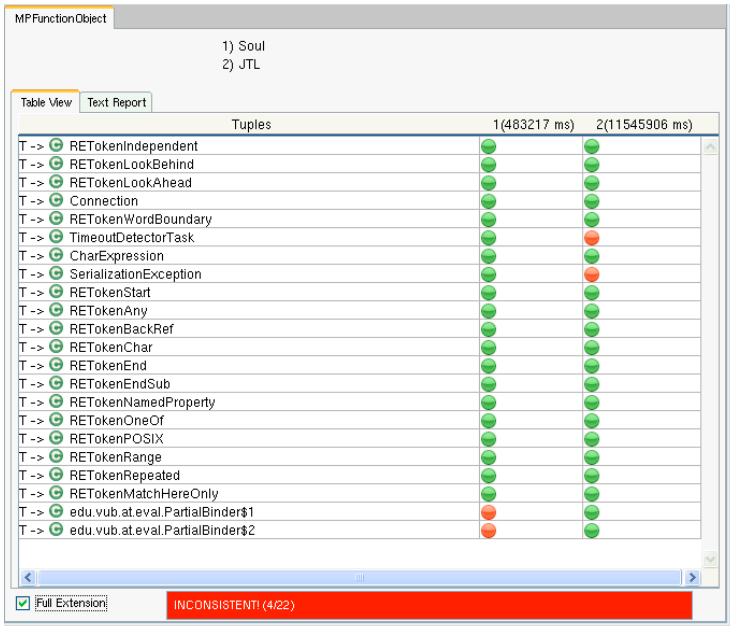


Figure 8.8: The Function Object μ-pattern in the AMBIENTTALK interpreter.

code of the program. Figure 8.8, for instance, depicts the results for the *Function Object* μ-pattern on the AMBIENTTALK interpreter. It describes concrete classes that have at least one field and a single public instance method (all declared or inherited, but not from `Object`). Instances identified by SOUL have a green entry in the second column. Instances identified by JTL have a green entry in the third column. The source code of `TimeOutDetectorTask` and `SerializationException` exhibits the characteristics of the Function Object μ-pattern. They are therefore identified by SOUL. However, they are not identified by JTL as they inherit instance methods from library classes `java.util.TimerTask` and `java.io.ObjectStreamException` respectively. On the other hand, SOUL did not identify the anonymous class declarations within `PartialBinder` because our example-based specification did not take such classes into account.

8.3 Detecting Bug Patterns

Of the pattern kinds introduced in Section 2.1, we also detected bug patterns to demonstrate the *general-purpose applicability* of our approach. We have concentrated on heterogeneously characterized bug patterns. The previous sections already demonstrated that our detection mechanism is able to detect implicit implementation variants of a pattern.

8.3.1 Detecting Potential Run-time Exceptions

In this section, we exemplify a code snippet that inadvertently raises a `NullPointerException` at run-time. The resulting specification illustrates the general-

```
if jtMethodDeclaration(?m,?interpretation){
    ?modList ?type ?name(?pList) {
      if(?x == null)
        ?x.?message(?aList);
    }
  },
  absolutelyNot(jtMethodDeclaration(?m,?interpretation){
                  ?modList ?type ?name(?pList) {
                    if(?x == null) {
                      ?x = ?exp;
                      ?x.?message(?aList);
                    }
                  }
                })
```

Tuples	1(2307 ms)
m -> ExamplesPEPMTest >> public nulltest2(Integer x) message -> floatValue	0.4
m -> ExamplesPEPMTest >> public nulltest(Integer x) message -> intValue	0.72

Figure 8.9: Detecting inadvertent invocations on null.

purpose nature of our prototype. It identifies code that raises such an exception at run-time, but is not flagged as suspicious by the compiler.

The JVM raises a `java.lang.NullPointerException` when "an application attempts to use `null` in a case where an object is required." This exception is, among others, raised when a message is invoked on the `null` value.

Figure 8.9 depicts a query that can be used to detect *inadvertent* invocations on the `null` value. The first condition exemplifies a method *?m* that sends a *?message* to a variable *?x* that is *guaranteed* to evaluate to `null` at run-time. This is because the invocation is located in the `true`-branch of an `if`-statement that checks whether the expression is `null`. The second condition ensures that the variable is not assigned in between the `null`-check and the invocation.

Figure 8.9 also depicts the solutions to this query, evaluated under the control flow interpretation against the following program:

```
public void nulltest(Integer x) {
  if(x == null)
    x.intValue();
}
public void nulltest2(Integer x) {
  if(x == null) {
    this.performOperation(x);
  }
}
private void performOperation(Integer y) {
  y.floatValue();
}
public void notNullTest(Integer x) {
  if(x == null) {
    x = new Integer(1);
    x.intValue();
  }
}
```

Note that method `nulltest2` is included in the solutions because it invokes

`floatValue` indirectly on a `null`-value.

Evaluation The example-based specification is reminiscent of the METAL and CONDATE specifications for `null` pointer dereferences in C depicted in Figure 3.9 and Figure 3.10 respectively. However, those specifications also take `null` values into account that stem from uninitialized variables. While specialized bug detection tools excel at the detection of common bug patterns such as null pointer dereferences, a tool supporting the detection of user-specified patterns can also be applied to application-specific bugs.

8.3.2 Detecting Design Pattern Implementation Pitfalls

In this section, we specify and detect in an example-driven manner common pitfalls in implementations of the *Singleton*, the *Observer* and the *Composite* design pattern.

Through these bug patterns, we will demonstrate that *both class-level and instance-level patterns can be specified in a uniform language*. Concretely, we will reuse the design pattern specifications of Section 8.1 to detect the *classes* that participate in a design pattern. We will exemplify the corresponding pitfalls at the instance-level: as *instances* of the participating classes that exhibit the characteristics of the pitfall.

1/ Singleton Implementations with Protected Constructors

The *Singleton* implementation in the Hannemann's base program [HK02], discussed in Section 8.1, has a constructor that is declared `protected` rather than `private`. This enables a subclass to initialize its inherited data members through a super constructor invocation. The pitfall is that `protected` constructors can also be invoked by other classes in the same package.

Rather than merely issuing a warning about all such constructors, the example-based specification depicted in Figure 8.10 looks for an expression *?exp* that creates an instance of a *?singleton* (line 5) that does not unify to any extent with the *?uniqueInstance* of the singleton. This expression is guaranteed to be in a *must-not-alias* relation with the unique instance of the singleton (i.e. their points-to sets are disjoint). The singleton itself is identified through the predicate defined in Figure 8.1. Lines 2–4 of the specification are therefore situated at the class-level, while lines 5–6 are situated at the instance-level.

The solutions at the bottom of the figure stem from method `test1` of Hannemann's base program [HK02]. As the method is defined in the same package as `PrinterSingleton`, it has access to the latter's `protected` constructor:

```
1  package ca.ubc.cs.spl.aspectPatterns.examples.singleton.java;
2  public class Main {
3    private static void test1() {
4      printer1 = new PrinterSingleton();
5      printer2 = new PrinterSingleton();
6      printer3 = new PrinterSingleton();
7      ...
8   }
9   ...
10 }
```

```
1  if ?interpretation equals: lexical,
2     ?singleton isSingletonClassForInstance: ?uniqueInstance
3                accessedThrough: ?instance
4                underInterpretation: ?interpretation,
5     jtExpression(?exp,?interpretation){new ?singleton(?argList)},
6     absolutelyNot(?exp equals: ?uniqueInstance),
7     ?parent equals: [?exp getParent]
```

Tuples	1(3935 ms)
singleton -> PrinterSingleton exp -> new PrinterSingleton() parent -> printer2=new PrinterSingleton()	0.81
singleton -> PrinterSingleton exp -> new PrinterSingleton() parent -> printer1=new PrinterSingleton()	0.81
singleton -> PrinterSingleton exp -> new PrinterSingleton() parent -> printer3=new PrinterSingleton()	0.81

Figure 8.10: Detecting an implementation pitfall of the *Singleton* design pattern.

2/ Lapsed Listeners in Observer Implementations

"Lapsed listeners" [Liv05] are observer participants in implementations of the *Observer* design pattern that are no longer needed, but never unregister from their subject. This is a problem for the subject participants identified by the specification in Figure 8.1. Their *?observers* field precludes an unneeded observer from being garbage collected.

The example-based specification in Figure 8.11 therefore identifies all *?observer* objects that are added to a *?subject* (lines 6–8), but not removed from it (lines 9–11). The final condition is optional. It identifies the expression that instantiated this observer object. Lines 2–5 of the specification are therefore situated at the class-level, while lines 6–14 are situated at the instance-level.

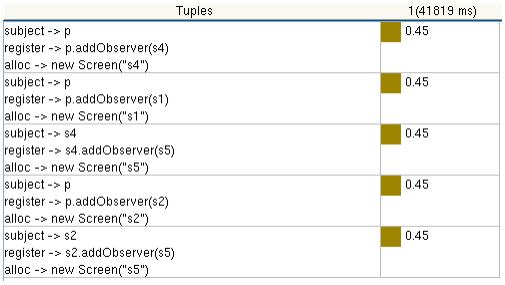
The solutions at the bottom of the figure stem from a modified method of the academic base program. The original version did not unregister a single observer from its subject. To this method, we added an expression "`p.removeObserver(s3)`". In contrast to all other observers, observer `s3` is therefore not included in the solutions at the bottom of the figure.

Note that the depicted specification only detects possible lapsed listeners. It does not identify the point in the program's execution after which an observer is no longer needed, nor does it specify that the *?unregister* expression should be executed after the *?register* expression. It can therefore only be used to issue warnings.

3/ Unvisited Components of a Composite

The *Visitor* and *Composite* design patterns are often used together [GHJV94]. This is, for instance, the case in the base program depicted in Figure 5.4. There, `Leaf1` and `Leaf2` are subclasses of `Component` and implement the *Composite* design pattern. Class `ComponentVisitor` is the abstract root of a hierarchy of visitors for the `Component` hierarchy. It defines methods `visitLeaf1(Component)` and `visitLeaf2(Component)`. Class `SumComponentVisitor` extends `ComponentVisitor` and overrides these methods. Class `Component` is implemented as follows:

```
if ?interpretation equals: lexical,
   ?subjectClass isSubjectOfObserver: ?observerClass
                 add: ?addObserver remove: ?removeObserver
                 notify: ?notifyObservers update: ?update
                 underInterpretation: ?interpretation,
   jtExpression(?register,?interpretation){
     ?subject.?addObserver(?observer)
   },
   absolutelyNot(jtExpression(?unregister,?interpretation){
                   ?subject.?removeObserver(?observer)
                 }),
   jtExpression(?alloc,?interpretation){
     ?observer := new ?observerClass(?argList)
   }
```

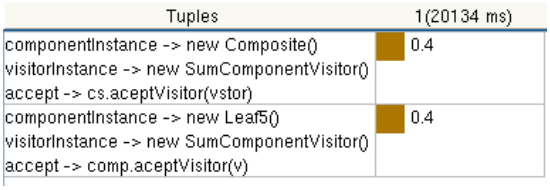


Tuples	1(41819 ms)
subject -> p register -> p.addObserver(s4) alloc -> new Screen("s4")	0.45
subject -> p register -> p.addObserver(s1) alloc -> new Screen("s1")	0.45
subject -> s4 register -> s4.addObserver(s5) alloc -> new Screen("s5")	0.45
subject -> p register -> p.addObserver(s2) alloc -> new Screen("s2")	0.45
subject -> s2 register -> s2.addObserver(s5) alloc -> new Screen("s5")	0.45

Figure 8.11: Detecting an implementation pitfall of the *Observer* design pattern.

```
if ?i equals: controlflow,
   ?composite isCompositeClassForComponentType: ?
              andLeafClass: ?leaf underInterpretation: ?i,
   or(?componentClass equals: ?composite,?componentClass equals: ?leaf),

   jtExpression(?componentInstance,?i){new ?componentClass(?arg2List)},
   ?visitorClass classDeclarationHasName: {.*ComponentVisitor},
   jtExpression(?visitorInstance,?i){new ?visitorClass(?arg1List)},
   ?acceptVisitor equals: {.*Visitor},
   jtExpression(?accept,?i){?componentInstance.?acceptVisitor(?visitor) },
   ?visitComponent equals: {visit.*},
   absolutelyNot(jtExpression(?visit,?i){
                   ?visitorInstance.?visitComponent(?componentInstance)
                 })
```



Tuples	1(20134 ms)
componentInstance -> new Composite() visitorInstance -> new SumComponentVisitor() accept -> cs.aceptVisitor(vstor)	0.4
componentInstance -> new Leaf5() visitorInstance -> new SumComponentVisitor() accept -> comp.aceptVisitor(v)	0.4

Figure 8.12: Detecting instances of *Composite* participants that are not visited by a *Visitor* instance.

```
1 public abstract class Component {
2  public void addComponent(Component element) {}
3  public void abstract aceptVisitor(ComponentVisitor v);
4 }
```

The class defines an abstract method `aceptVisitor`.[11]. This method is overridden in `Leaf1` and `Leaf2` such that the double dispatching idiom, which is at the heart of the *Visitor* pattern, is implemented properly.

A common pitfall of such combinations of a `Visitor` and a `Composite` is that one of the components does not properly implement the method that accepts the visitor. For instance, because the method was implemented with an empty body (i.e. with the intent to provide a proper implementation as soon as the program compiles).

The query depicted in Figure 8.12 identifies instances of a *?componentClass* on which the double dispatching idiom is initiated, but not completed. Lines 2–4 identify the classes that participate in the *Composite* implementation through the specification depicted in Figure 8.2. They are situated at the class-level. Lines 5–13 are situated at the instance-level. Lines 5–9 identify a *?componentInstance* and a *?visitorInstance* such that the former accepts the latter through an invocation "*?componentInstance*.*?acceptVisitor*(*?visitor*)". This invocation initiates the double dispatching protocol. Lines 11–13 only succeed when there is no corresponding "*?visitorInstance*.*?visitComponent*(*?componentInstance*)" that completes the protocol. Note that lines 8 and 9 use heuristics on *?acceptVisitor* and *?visitComponent* to restrict the corresponding invocations to those that are part of the prototypical *Visitor* implementation.

We evaluated the query against a modified version of the base program depicted in Figure 5.4. In this version, a `Composite` instance is configured with instances of `Leaf1`, `Leaf2` and `Leaf5` —of which the latter class implements method `aceptVisitor` with an empty body:

```
1 Composite cs = new Composite();
2 cs.addComponent(new Leaf1());
3 cs.addComponent(new Leaf2());
4 cs.addComponent(new Leaf5());
5 SumComponentVisitor vstor = new SumComponentVisitor();
6 cs.aceptVisitor(vstor);
```

The solutions for this base program are depicted at the bottom of Figure 8.12. Included are the instance of `Leaf5` that is instantiated on line 4 above, but also the instance of `Composite` that is instantiated on line 1. The latter instance is included because it forwards the `aceptVisitor` message to its leafs (cf. Figure 5.4).

Note that the query in Figure 8.12 does not verify whether method *?visitComponent* is named after the component that accepts the visitor. This is not the case, for instance, when an implementation is inherited. This can be specified easily in terms of structural rather than behavioral characteristics.

Evaluation The above specifications re-use the specifications of Section 8.1 to detect the *classes* that participate in a design pattern. The pitfalls themselves are exemplified at the instance-level: as *instances* of the participating classes that exhibit the characteristics of the pitfall. This way, we demonstrated the facilities for reuse and abstraction of our specification language well as well as its support for

[11] The spelling error is deliberate

specifying behavioral and non-behavioral pattern characteristics in a uniform language.

8.4 Guidelines for Exemplifying a Software Pattern

Having presented example-based specifications for various software patterns, we briefly present some rough guidelines for developers to follow when exemplifying other software patterns.

One should first consider whether the pattern in question is situated primarily at the instance-level or at the class-level. In the latter case, a combination of `jtClassDeclaration/2` and `jtInterfaceDeclaration/2` template terms is generally in order.

In the former case, a combination of `jtExpression/2` template terms is in order. If the evaluation order of these expressions is important, they can be grouped in a `jtMethodDeclaration/2` template term. However, our research prototype resolves such terms in a time-consuming manner under the control flow interpretation. It is therefore best to restrict the candidate matches for method declaration template terms as early as possible. For instance, by exemplifying statements in its body that are matched intra-procedurally rather than inter-procedurally. Or by exemplifying its formal parameters, its return type, its modifiers, the class in which it is defined etc...

In either case, the template terms themselves should exemplify the prototypical implementation of a pattern's *essential* characteristics. The detection mechanism of our approach is geared towards finding variants of this implementation. Matches for a template term have to exhibit all exemplified characteristics, but may exhibit additional ones. Exemplifying unessential characteristics should be avoided as these needlessly constrain the implementation variations that will be recalled. The bindings for a term's variables can always be constrained further outside the term. For instance, through logic conditions or Smalltalk terms. Finally, dividing the concrete syntax of the implementation over multiple template terms allows for finer-grained control over the matches (cf. Section 7.3).

Finally, multiple prototypical implementations of a pattern can be enumerated in a logic disjunction. Each disjunct can also be used as the body for a rule that implements a common predicate. This allows weighting the rules with different truth degrees that express the confidence in the prototypical implementation they exemplify.

8.5 Concluding Evaluation

In Section 2.6, we formulated the criteria for a general-purpose pattern detection tool. Such a tool is not specialized in a particular application of pattern detection, but can be applied to any of the software engineering problems enumerated in Section 2.3. Table 2.1 summarizes these criteria.

Our example-driven approach to pattern detection fulfills all of the criteria for a general-purpose pattern detection tool. In Chapter 4, we *motivated the cornerstones of our approach in terms of their contributions to these criteria.* These are summarized in Table 4.1.

We conclude this chapter by evaluating our approach as a whole on these criteria using the design patterns, μ-patterns and bug patterns exemplified above.

8.5.1 Evaluation on the Criteria for the Pattern Specification Language

Our approach supports specifying behavioral and non-behavioral pattern characteristics in a uniform language (criterion **CSL1**). We specified all patterns in this chapter using example-based specifications. Most of these patterns are heterogeneously characterized.

Consider the specification for the *Observer* pattern in Figure 8.1. It exemplifies the *?subjectClass* participant as a class with at least a field and three methods. These are structural characteristics. The same goes for the formal parameters of methods *?addObserver* and *?removeObserver*. Both are required to have a single parameter of the same type *?observerType*. Method *?notifyObservers* is exemplified as a method with two successively evaluated instructions. The first evaluates to the *?observers* field and the second invokes a method on an *?observer* that has been added to this field through method *?addObserver*. Clearly, these are control flow and data flow characteristics. The *lapsed listener* implementation pitfall of the *Observer* pattern augments the above characteristics with instance-level data flow characteristics.

In Section 7.4, to conclude, we moreover exemplified the syntactic and data flow characteristics of potentially enhanceable `for`-statements, the structural characteristics of application-specific coding conventions, and the control flow and data flow characteristics of the protocol an API expects to be adhered to.

The example-based specifications presented in this chapter are descriptive specifications of a pattern's characteristics, rather than an operational implementation of the search for its instances (criterion **CSL2**). Most specifications resemble actual source code excerpts. For instance, the *Template Method* in Figure 8.1 or the one for the bug pattern that describes inadvertent invocations on `null` in Figure 8.9. However, we encountered problems specifying cardinality constraints such as "for-all" and "at-least-one" using template terms alone. In the specification for the *Function Pointer* in Figure 8.6, for instance, we had to resort to complicated idioms of non-native operators that detract from its descriptiveness. These would have been easier to express using the higher-order predicates of logic meta programming. The same is true, but to a lesser extent, for the sub-typing relation between two types. The example-based specification for the *Decorator* design pattern [GHJV94], for instance, requires a disjunction of several template terms to exemplify the different manners in which the decorator participant can reside in the sub-type hierarchy of another type (cf. lines 7–24 of Figure 8.2). The relational nature of logic programming facilitates quantifying over the reification predicates for structural information to express such structural characteristics. We did this, for instance, on lines 12–13 of the *Composite* specification in Figure 8.2. This specification illustrates that template terms can be combined with regular logic terms to alleviate these problems.

Our specification language as a whole supports expressing explicit points of variation among pattern instances in different ways (criterion **CSL3**). We already mentioned the disjunctions of template terms in the specification for the *Decorator* pattern. The specification for the *Singleton* in Figure 8.1 illustrates that different template terms (each specifying a different prototypical implementation of the pattern) can also implement the same predicate. In the specification for the *Decorator* pattern, we moreover illustrated that users can associate different weights with each exemplified prototypical implementation.

Predicate definition is also the primary means for abstraction and reuse among specifications (criterion **CSL4**). This is illustrated by the specifications for the design pattern implementation pitfalls in Section 8.3.2.

Finally, none of the specifications in this chapter expose details of the program analyses that enable detecting implicit implementation variants of behavioral characteristics (criterion **CSL5**) that are included in our program representation (criterion **CPR1**).

8.5.2 Evaluation on the Criteria for the Pattern Detection Mechanism

All of the reported pattern instances stem from the source code of the base program (criterion **CDM1**). In fact, this allowed us to detect errors in the JTL specifications for the μ-patterns. To the original JTL specification for the *Function Pointer* μ-pattern, for instance, we had to add additional `!synthetic` goals such that its instances can have synthetic members which are present in the bytecode, but not in the source code.

The truth degrees associated with each reported pattern instance facilitate their user assessment (**CDM2**). For the *Decorator* instances, for example, we were able to derive whether a method invocation and method declaration unified according to the dynamic or static type of the former's receiver. However, the overall ranking could be improved (cf. Section 9.4.2). Specifically, instances that required two unifications that could introduce false positives are not ranked lower than instances that required only one. This is a limitation of our instantiation of the fuzzy logic cornerstone, not of the cornerstone itself. We will address this shortcoming in future work.

That our detection mechanism recognizes implicit points of variation among pattern instances (criterion **CDM3**), is perhaps best illustrated by the *Observer* specification through the occurrences of *?observer* in the *?notifyObservers* and *?addObserver* method. In the academic base program [HK02], the former is bound to an expression ‘‘`((ChangeObserver)e.next())`’’, while the latter is bound to a formal parameter “`o`”. Effectively, these occurrences express that at least one *?observer* added through method *?addObserver* should be notified upon a change in the subject.

Although our open implementation cornerstone enables defining additional example-based interpretations for template terms as well as additional domain-specific unification extensions (**CDM4**), this was not required for the patterns in this chapter.

CHAPTER 9

CONCLUSION AND FUTURE WORK

This chapter concludes our discourse on the logic meta programming foundation for example-driven pattern detection presented in this dissertation. Before enumerating some interesting directions for future research, we revisit the problem statement and restate the contributions of our dissertation.

9.1 Problem Statement Revisited

Pattern detection tools that support user-specified characteristics have valuable applications throughout the development process. However, specifying a pattern's characteristics and subsequently assessing the reported instances is hard. Such characteristics not only concern the structure of a program, but also the execution order of its instructions and the values operated on by these instructions.

As many patterns are heterogeneously characterized, we identified the need for a specification language in which behavioral as well as non-behavioral characteristics can be specified uniformly. We moreover identified the need for a detection mechanism that recognizes implicit implementation variants of behavioral characteristics (i.e. those implied by the semantics of the programming language). This precludes users from having to enumerate each variant in a specification. Recent advances in program analysis have enabled detecting these variants in industrially-sized programs. However, different analyses implement different trade-offs with respect to precision and analysis time. Assessing the extent to which a reported pattern instance exhibits the specified characteristics therefore requires detailed knowledge about a tool's enabling analyses. There is therefore also a need to facilitate user assessment of the identified pattern instances.

In response to these problems, we formulated criteria for each of the dimensions in the design of a pattern detection tool: its specification language, its detection mechanism and its program representation. When fulfilled, these criteria result in a *general-purpose* pattern detection tool that can be applied throughout the development process to detect *behavioral and non-behavioral* pattern characteristics using *descriptive and declarative* specifications in a *uniform* language.

In an extensive survey, we found that the state of the art in pattern detection

tools is currently lacking with respect to these criteria. For instance, we were able to structure this survey according to the characteristics each tool is primarily intended for. This confirms the need for a general-purpose tool. One can object that non-syntactic characteristics can be expressed in terms of syntactic characteristics if the specification language is sufficiently powerful. We argued that such specifications are far from descriptive, have recurring parts and possibly lead to a lower recall and false positives. In particular for data flow and control flow characteristics, the semantics of the programming language must be taken into account correctly.

9.2 Conclusion

In this dissertation, we presented an example-driven approach to pattern detection that fulfills all of the criteria for a general-purpose pattern detection tool. Its specification language enables exemplifying a pattern through code excerpts that correspond to the prototypical implementation of its essential characteristics. Moreover, its detection mechanism recognizes implicit points of variation among the pattern's instances (i.e. those implied by the semantics of the programming language).

We presented this approach in terms of five cornerstones and their interdependencies:

Cornerstone: logic meta programming enables specifying a pattern's characteristics as an expressive logic formula. To identify the program elements that exhibit the pattern's characteristics, this formula has to quantify over a reified program representation. As the founding cornerstone, LMP also lends our detection mechanism its proof procedure for such formulas.

Cornerstone: example-based specification enables exemplifying the prototypical implementation of a pattern's essential characteristics. Within logic formulas, this implementation can be exemplified as a code excerpt in the concrete syntax of the program under investigation —augmented with logic variables to indicate points of variation. This obviates the need to quantify explicitly over the reified program representation to express such characteristics. Developers are therefore shielded from the details of the program representation and its reification.

Cornerstone: domain-specific unification ensures that variable bindings are consistent across the logic conditions and the code excerpts in a logic formula. It incorporates whole-program analyses to determine whether reified program elements implement the same pattern characteristic. This obviates the need to enumerate these variants in a specification. Moreover, it hides the intricate details of the analyses that enable their detection.

Cornerstone: fuzzy logic ensures that each reported pattern instance is quantified by the extent to which it exhibits the characteristics in a specification. The smaller this extent, the more likely the instance is a false positive. This ranking facilitates assessing a large amount of results. False positives may result from imprecision in the enabling analyses of the domain-specific unification procedure. Under this procedure, unifying two reified program elements can succeed where the general-purpose procedure fails.

Cornerstone: open implementation crosscuts the implementations of the other cornerstone. Each cornerstone provides a meta-interface through which it can be extended at a higher level of abstraction than its implementation.

We discussed each cornerstone as instantiated in the research prototype that we used to validate our approach. We identified patterns that are representative for each kind of pattern characteristic and successively specified these as a logic formula, a fuzzy logic formula with domain-specific unification and as a logic formula containing source code excerpts. This clarified how each cornerstone contributes to the criteria for a general-purpose pattern detection tool.

Finally, we evaluated our approach as a whole on these criteria by detecting several patterns: design patterns, μ-patterns and bug patterns ranging from common pitfalls in the implementation of design patterns to inadvertent invocations on `null`.

9.3 Contributions Restated

The following summarizes the contributions of each chapter:

- In Chapter 2, we formulated the *criteria for a general-purpose pattern detection tool* and introduced sufficient background to motivate each criterion. We identified the key dimensions in the design of a tool that supports detecting user-specified patterns. We surveyed the applications of pattern detection in software engineering and identified the different kinds of characteristics each requires identifying: syntactic, structural, control flow and data flow characteristics. *For each kind of pattern characteristic, we investigated which configurations in the design of a pattern detection tool support its detection.* In particular, we emphasized the intricacies of the analyses that enable detecting implementation variants of control flow and data flow characteristics.

- In Chapter 3, we presented an *extensive survey* of the different specification languages, detection mechanisms and program representations used by the state of the art in pattern detection tools. Moreover, we evaluated each tool on the aforementioned criteria.

- In Chapter 4, we *introduced and carefully motivated the cornerstones of our approach* with respect to the criteria for a general-purpose pattern detection tool. We emphasized the *quantification-related and unification-related shortcomings* of logic meta programming in a pattern detection setting. These are remedied by the example-based specification and domain-specific unification cornerstones respectively.

- In Chapter 5, we instantiated the logic meta programming cornerstone. This instantiation consists of SOUL and the CAVA *library for reasoning about Java programs.* The latter constitutes a technical contribution of our dissertation. Unique is its *identity-based reification to objects*: the reified version of an AST node is the AST node itself (i.e. an `org.eclipse.jdt.core.dom.ASTNode` instance). This way, reconstructing the actual AST node from its reified counterpart is trivial at any point in the proof procedure (e.g. in the unification procedure). We established a *linguistic symbiosis* between SOUL and Java to implement this reification.

- In Chapter 6, we instantiated the fuzzy logic and domain-specific unification cornerstones. This instantiation consists of a *fuzzy version of* SOUL with *domain-specific extensions* to the general-purpose unification procedure. We quantify the solutions to a logic goal with truth degrees. The logic rules used in the resolution of a goal determine the upper bound for these degrees. A solution can be ranked lower than the other solutions identified by the same rules. This is the case if it requires a domain-specific unification that could introduce false positives.

  To support the natural use of unification to quantify over AST nodes, *reified AST nodes unify with structurally equivalent compound terms* —even if the reified version of an AST node is not a compound term. Moreover, *reified AST nodes unify if they represent different implementations of the same characteristic*. To this end, the domain-specific unification extensions consult the results of whole-program analyses: a semantic analysis, an inter-procedural points-to analysis and an intra-procedural must-alias analysis. Note that these were not reified by the CAVA library to shield users from their intricate details.

- In Chapter 7, we instantiated the example-based specification cornerstone as *template terms* in the fuzzy version of SOUL. These are resolved by matching their source code excerpt against the program representation. Matches should exhibit the characteristics exemplified by the code excerpt of the template term. However, a single code excerpt can exemplify different pattern characteristics. *An AST node therefore always matches a template term under a particular example-based interpretation of the excerpt.* We defined three standard interpretations: the syntactic, lexical and control flow interpretation. *The points of variation among the matches for a template term differ under each interpretation.* Under the control flow interpretation, for instance, the control flow characteristics of the source code excerpt exemplify the intended matches. *The translational semantics of additional interpretations can be specified through logic rules.*

- In Chapter 8, we validated our approach on the criteria for a general-purpose pattern detection tool by exemplifying and subsequently assessing the reported instances for several software patterns.

9.4 Future Work

In future work, we should address the technical limitations of our research prototype. We discussed these limitations at the end of each chapter that instantiates a cornerstones of our approach. We recall the most important ones:

- First of all, we should alter the translational semantics of the control flow interpretation to properly support universal and existential path queries with complements. Specifically, we should bypass the control flow graph traversal predicates of the CAVA library and their limitations (cf. Section 5.5.2). For instance, by adopting one of the algorithms for evaluating path queries surveyed in Section 3.4.3.

- We should also address the performance disadvantage of SOUL with respect to traditional Prolog implementations (cf. Section 5.5.1). There is ample

of room for optimization. For instance, by using a Warren Abstract Machine [War83] rather than a completely interpreted evaluator. Incorporating tabled resolution [RC97, CW96] would not only obviate the need to cache the results of strategic predicates manually, but also allow left-recursion in the Definite Clause Grammar for template terms (cf. Section 7.6).

- Finally, we should delay combining unification degrees with resolution degrees until a truth degree is requested (cf. Section 6.8). This would rank a solution that required two unifications that could introduce false positives lower than a solution that only required one. For instance, by adopting a unification procedure in the style of LIKELOG [AF99]. Adopting multiplication to quantify conjunction would address this limitation as well, but would lead to small truth degrees if the same condition is repeated in a query.

The remainder of this section discusses the open research questions related to our approach rather than the technical limitations of its instantiation.

9.4.1 Exploring Example-Based UML Diagrams

Our example-based specifications for Gil and Maman [GM05]'s μ-patterns illustrate that cardinality constraints such as "for-all" and "at-least-one" are not easily specified using template terms alone (cf. Section 8.2).

The same is true for structural pattern characteristics such as the sub-typing relation between two types. The example-based specification for the *Decorator* design pattern [GHJV94], for instance, requires a disjunction of several template terms to exemplify the different manners in which the decorator participant can reside in the sub-type hierarchy of another type (cf. lines 7–24 of Figure 8.2). The participant can either extend the class (or a sub-class thereof) that declares this type, implement the interface (or a sub-interface thereof) that declares this type, or extend a class (or a sub-class thereof) that implements an interface (or a sub-interface thereof) that declares this type.

The relational nature of logic programming facilitates quantifying over the reification predicates for structural information to express such structural characteristics. Higher-order predicates can moreover be used to specify cardinality constraints. However, we have already begun an initial exploration of how such characteristics can be specified in a more straightforward manner.

Figure 9.1 depicts two diagrams that exemplify the prototypical Smalltalk implementations of the *Composite* and *Visitor* patterns. We are investigating a UML-like notation extended with template terms as well as regular logic terms. The diagram at the right, for instance, requires that all *?element* classes in the sub-type hierarchy of `AbstractTerm` that have a method *?acceptMethod* double dispatch to method *?visitMethod* of `SimpleTermVisitor`. It also expresses that there should be at least one such *?element*. The diagram exemplifies the sub-type relation in UML notation. The diagram at the left states that a *?composite* should be in a UML composition relation with its component type *?comp*. It uses the UML notation for this relation rather than exemplifying the *?composite* with a *?field*. This allows this relation to be implemented in different ways.

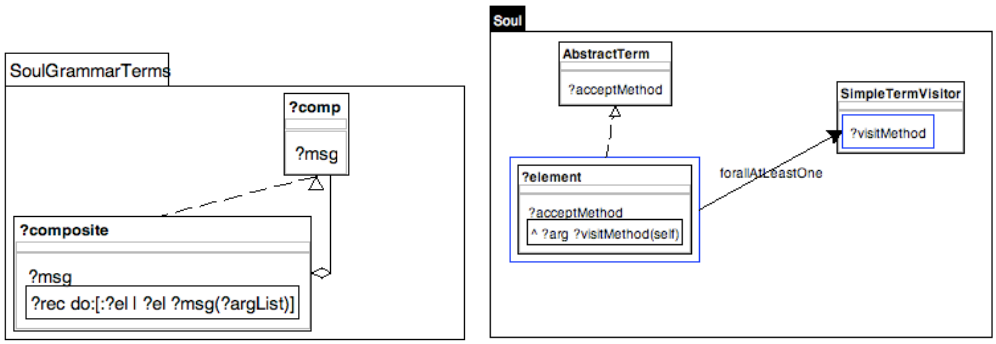


Figure 9.1: Exploring example-based diagrams for *Composite* and *Visitor*.

9.4.2 Refining Ranking of Results

Section 6.4 deduces a unification degree for each domain-specific extension to the unification procedure. In our prototype, only the unification based on points-to analysis can introduce false positives. The associated unification degree is fixed for all program elements that are in a may-alias relation. It immediately halves the truth degree computed by a resolution without unification degrees. This is an effective, but crude measure to ensure that these solutions stand out. *In future work, we want to investigate how to let the unification degree vary among all elements that are in a may-alias relation.* For instance, by using a probabilistic points-to analysis [SS06] which computes the probability that a reference points to a heap location.

If multiple domain-specific extensions can introduce false positives (e.g. by using different whole-program analyses to recognize implicit points of variation), the ranking among their unification degrees should be established in a methodological manner. For instance, based on empirical experiments that assess their relative precision.

Section 7.2.5 deduces truth degrees for each example-based interpretation. Solutions to a template term are ranked according to the extent to which they exhibit the characteristics in a specification. The smaller this extent, the more likely a reported instance is a false positive. For user-defined interpretations, a ranking that facilitates user assessment of results (criterion **CDM2**) cannot be guaranteed.[1] Users should ensure that their truth degrees are compatible with the predefined ones. If necessary, the open implementation cornerstone allows changing the predefined truth degrees. These degrees annotate the logic rules that implement the translational semantics of each example-based interpretation. To facilitate changes, the rules could be annotated with Smalltalk terms that retrieve truth degrees from a central configuration point

In future work, we want to investigate how a suitable ranking can be deduced automatically for a given set of domain-specific extensions to the unification procedure and a given set of example-based interpretations for template terms. Their historical precision, user feedback and application-specifics could be used as input to machine-learning algorithms that evolve the ranking over time.

[1] The same goes for LMP specifications with user-defined fuzzy predicates.

9.4.3 Integrating Context-Sensitive Points-to Analyses

The points-to analysis in our program representation is context-insensitive. It does not parametrize the points-to sets for expressions in a method with a static representation of the method's possible run-time invocation contexts. Adopting a context-sensitive points-to analysis would enable deriving more precise may-alias information.

Section 6.6.4 gave an example of a false positive introduced by the domain-specific unification extension that relies on may-alias information. More precise points-to sets would have eliminated this false positive. Context-sensitive points-to analysis results can be accessed without having to provide a static representation of an invocation context (e.g. the points-to sets of the topmost receivers on the call stack). To access the context-sensitive points-to sets for two expressions, the domain-specific unification extension would not have to be changed. The retrieved points-to sets are more precise because the analysis analyzes different invocations of all methods in the program separately.

In its inter-procedural search for instructions that are executed in the exemplified order, the control flow interpretation of a template term simulates a call stack. In future work, we therefore want to explore how the contents of this simulated stack can be accessed from within the unification procedure. This would allow accessing the points-to set for an expression within a particular invocation context of the method it resides in.

In our current prototype, this is already possible in an ad-hoc manner because we implemented the simulated stack as a global Smalltalk object that is accessible through linguistic symbiosis. However, this has proven a bad implementation choice because splits in the control flow require restoring this stack to a previous state when the control flow traversal predicates of CAVA are backtracked over. Managing global Smalltalk state within a SOUL program is difficult in general.

SOURCES OF BASE PROGRAM INFORMATION

Section 2.5 discusses the different kinds of program information that are necessary to support each kind of pattern characteristic. This appendix complements Section 2.5 in that it discusses how to obtain this information about the program.

A.1 Obtaining Syntactic Information

Abstract syntax trees encode the syntactic relations between a program's constructs. They are either construted from the derivation trees computed by a *syntactical analysis* of the program's concrete syntax or constructed immediately during the analysis itself. Syntactical analysis has been studied extensively in computer science. We refer the reader to Aho et al. [AU72] for a comprehensive treatise of the theory of grammars in computer science and to the more recent Grune et al. [GJ90] for a treatise of the many parsing algorithms that have been developed over the years. We refer the reader to Klint et al. [KLV05] for a discussion of the important engineering aspects of grammar-related software such as the software used to obtain abstract syntax tree representations for pattern detection tools.

Declarative Syntax Specifications

Many parser generator tools follow the example set by the ubiquitous Unix tool *Yacc* [Joh79, LMB92] in which imperative code adorning concrete syntax grammar rules is responsible for constructing an abstract syntax tree.

However, declarative specifications of both concrete and abstract syntax are possible. This is demonstrated, for instance, by the *Syntax Definition Formalism* (*SDF*) [SDF08, VS00, HHKR89]. Here, constructors for abstract syntax terms adorn the grammar rules. The latter can moreover be specified in a modular manner. Programming language ambiguities are naturally supported both by generating a generalized parser (which allows forests of derivation trees) and by providing declarative disambiguation declarations (to prune the latter). Mostly due to its modularity and seamless integration of lexical analysis, *SDF* has been applied successfully in

parsing incomplete or erroneous legacy artifacts according to an island grammar [Moo01]. In this context, we will revisit *SDF* in our discussion of the extraction of structural program representations in Section 3.3.

In keeping with the potential of declarative programming, the problem solving strategy of Prolog renders a mere (DCG) [PW80] in both a recognizer and generator of the sentences it describes. The body of a DCG rule comprises a grammar's non-terminals and terminals in addition to regular logic predicates which can be used to incorporate semantics in the parsing process. A simple translation scheme maps DCG rules to regular logic rules over a token sequence. Backtracking naturally supports grammar ambiguities with infinite look-ahead, which may result in a forest of parse trees when the solutions are equally correct. Definite Clause Grammars have therefore enjoyed quite some popularity in the natural language processing community [PS02, BS02], but can be applied to programming languages as well [CH87]. The application of higher-order programming techniques [Nai96, CKW93] results in concise grammar specifications. Prolog's depth-first problem solving strategy, resolution [Rob65, EK76], does cause termination problems with left-recursive grammars, but these can either be avoided or explicitly supported by using a tabled variant of resolution [RC97, CW96] which results in an algorithm similar to chart parsing [PW83, XSB07]. As a general-purpose programming language, regular Prolog moreover supports different parsing algorithms for a logic grammar by means of a meta-interpreter interpreting its clauses or alternative translations schemes [CH87, BS02]. Finally, Lämmel et al. [LR01] rely on Prolog's operational interpretation and impure I/O predicates to interleave scanning and parsing of text according to a grammar expressed as plain Prolog clauses rather than DCG clauses.

A.2 Obtaining Structural Information

In Section 2.5.2, the Smalltalk run-time environment and the Java Model component of the Eclipse Java Development Toolkit (JDT) [Ecl08a] are listed as sources of structural program information. Apart from these, external tools can be used to extract such information from ASTs or compiled program artifacts. FAMIX [TDDN00], for instance, is a language-independent structural program representation that is shared by many reverse engineering and visualisation tools.

The ASF+SDF meta-environment, comprising the aforementioned declarative Syntax Definition Formalism (SDF) [SDF08, VS00, HHKR89] and its companion term rewriting language Algebraic Specification Language (ASF) [vdBKV07, vdBHKO02, Ber89, DHK96], have been used to extract structural program information for many program query languages (e.g. the relational RSCRIPT [Kli03]). ASF rules rewrite low-level terms recognized by an SDF-generated parser into the necessary structural program information.

A.3 Obtaining Control Flow Information

As argued in Section 2.5.3, deriving precise control flow information is a complex process. However, most compilers and static analysis frameworks have control flow graphs for function and method bodies readily available. The operations in these graphs usually stem from an intermediate program representation. Alternatively, a pattern detection tool can construct control flow graphs itself from a compiled,

intermediate or AST representation. Control flow edges can be added lazily on top of such a representation.

Constructing Whole-Program Control Flow Graphs for Higher-Order and Object-Oriented Programs

Section 2.5.3 described a relatively straightforward inductive process for constructing control flow graphs. However, this process is not applicable to languages that support function pointers or first-class functions. To determine the program's control flow, knowledge is required about the possible values expressions can assume. Traditionally, such information is delivered by data flow analyses. This introduces a circular dependency as such analyses require a control flow graph themselves to propagate results through. Specialized higher-order control flow analyses cope by intertwining both. We refer the interested reader to Shivers' [Shi04] for an overview.

Even for statically-typed object-oriented programs without closures, a similar circular dependency between control and data flow information can still be introduced by polymorphism and late binding. To determine at compile-time which method is invoked by a method invocation, knowledge is required about the values its receiver can assume. However, an imprecise control flow graph can be constructed using information about the invocation's signature, the receiver's declared type, the program's class hierarchy and the classes explicitly referred to in object instantiations (e.g. class hierarchy analysis [DGC95] and rapid type analysis [Bac97]). This initial control flow graph can subsequently bootstrap a data flow analysis which computes information about the heap objects the receiver might point to at run-time. The statically derived information about their run-time types can be used to further prune the call graph.

Constructing Whole-Program Control Flow Graphs in the Presence of Reflection

Constructing a complete call graph that comprises all of a program's reachable methods is still the subject of ongoing research. These should include methods that are invoked by the program's run-time environment (e.g. by native methods in Java), through reflection or remotely by another program. Often, consulting the user for assistance cannot be avoided. Recently, Livshits et al. [Liv06, LWL05b] have made advances in the automatic resolution of reflective calls. A points-to analysis allows them to discern two kinds of strings in reflective calls: those that originate from constants internal to the program and those that originate from external sources. While the former can be resolved completely, the latter require information provided by the user. Livshits et al. [Liv06, LWL05b] also demonstrate how user-involvement can be minimized by localizing idiomatic uses of reflection. In Java, it is for instance common for casts to follow invocations of the `Class.newInstance` method which returns an instance of static type `Object`. The type the object is cast to, when assumed to be successful, provides information about its possible dynamic type. Here, pattern detection not only relies on but also assists in program analysis.

A.4 Obtaining Data Flow Information

Data flow information is readily available from many compiler construction and program analysis frameworks. SOOT [VRCG+99] and JOEQ [Wha03] for Java both feature a substantive collection of analyses —including the reaching definitions [NNH05] and points-to analyses [Hin01] which are relied upon by the pattern detection tools surveyed in Section 3.5. Of the latter, several implementations are available for Java (e.g. [RMR01, LH03, WL04, Lho06, LL07]), each exploring different trade-offs between analysis precision and cost. We refer the interested reader to Ryder [Ryd03] for an overview of the dimensions of precision in the design of a points-to analysis and to Lhoták [LH06] for some empirical results.

APPENDIX B

ADDITIONAL VALIDATION-RELATED INFORMATION

This appendix complements Chapter 8 in which we validated our approach. We list statistics about the size of the base programs used throughout the chapter and present the example-based specifications for the μ-patterns [GM05] that we have not yet discussed.

B.1 Base Program Statistics

The following table lists some statistics about the size of the base programs we have used in the validation chapter:

	Hannemann's [HK02]	JHOTDRAW 5.1[jHo07]	AMBIENTTALK 2008/02/01 [Amb]
compilation units	104	133	250
type declarations	104	143	287
methods	273	1222	2290
statements	980	4794	9764
expressions	2991	15462	29628
reference expressions	2101	7280	19661

B.2 Undiscussed μ-Pattern Specifications

Table B.2 describes the μ-patterns [GM05]in natural language. Table B.1 compares the μ-pattern instances identified by SOUL and JTL in a compact manner. The first column lists the number of inconsistencies between both sets of instances, while the second column lists the size of the union of both sets. Most differences are due to the fact that JTL analyzes the bytecode of the program and the libraries it relies on, while our approach only analyzes the source code of the program (cf.

	Number of Inconsistencies	Size of the Union of Both Solution Sets
Outline	9 (JTL Exception)	9
Pseudo Class	1	3
Pure Type	0	64
State Machine	0	29
Trait	1	8
Restricted Creation	0	9
Sampler	0	3
Data Manager	0	0
Record	1	7
Sink	45 (JTL Exception)	45
Cobol Like	0	0
Function Object	4	22
Function Pointer	0	0
Common State	0	1
Immutable	103 (JTL Exception)	103
Stateless	18	34
Designator	0	0
Joiner	0	0
Pool	0	3
Taxonomy	0	12
Extender	36	115
Overrider	45	58
Box	0	0
Canopy	43 (JTL Exception)	43
Compound Box	1	8

Table B.1: Comparison of μ-patterns identified by SOUL and JTL.

Section 8.2.2). Note that JTL raised an exception on some of the μ-patterns. The example-based specifications evaluated by SOUL are listed below:

```
?class isBoxUnderInterpretation: ?interp if
  jtClassDeclaration(?class,?interp){
    class ?className *{
      !static ?type ?field = ?init;
      ?modList ?methodType ?methodName(?paramList) {
        ?field = ?assignedValue;
      }
    }
  },
  absolutelyNot(jtClassDeclaration(?class,?interp){
                  class ?className *{
                    !static ?otherType ?otherField = ?otherInit;
                  }
                },
                absolutelyNot(?otherField equals: ?field))

?class isCanopyUnderInterpretation: ?interp if
  jtClassDeclaration(?class,?interp){
    class ?className *{
      !static ?type ?field = ?init;
      ![?modList ?methodType ?methodName(?paramList) {
        ?field = ?assignedValue;
      }]
    }
  },
  absolutelyNot(jtClassDeclaration(?class,?interp){
                  class ?className *{
                    !static ?otherType ?otherField = ?otherInit;
                  }
                },
                absolutelyNot(?otherField equals: ?field))

?class isPseudoClassUnderInterpretation: ?interp if
```

μ-Pattern	Description
Outline	Abstract class of which a declared method (different from `main`) invokes an abstract method of the same class (declared or inherited).
Pseudo Class	Abstract class without instance fields and only abstract instance methods (all declared or inherited).
Pure Type	Abstract class or interface without fields, without concrete methods and at least one abstract method (all declared or inherited)
State Machine	Interface of which all declared non-private instance methods are parameter-less (and that has at least one of those methods).
Trait	Abstract class without instance fields, at least one abstract and at least one concrete method (all declared or inherited).
Restricted Creation	Class without declared public constructors and at least one declared static field of the same type as the class.
Sampler	Class with a declared public constructor and at least one declared static field of the same type as the class.
Data Manager	Class where all non-private instance methods are setters or getters (all declared or inherited).
Record	Concrete class with at least one public instance field, without private instance fields and without methods (all declared or inherited, but not from `Object`).
Sink	Class of which no declared method invokes another method.
Cobol Like	Class with a single static method and no instance fields or methods (all declared or inherited, but not from `Object`)
Function Object	Concrete class with at least one field and a single public instance method (all declared or inherited, but not from `Object`)
Function Pointer	Concrete class without field and a single public instance method (all declared or inherited, but not from `Object`)
Common State	Class with at least one declared non-final static field and without non-static fields or methods (all declared or inherited, but not from `Object`).
Immutable	Class in which all instance fields are private (and that has at least one such field) and that has no mutators for its instance fields (all declared or inherited).
Stateless	Class in which all fields are static and final (all declared or inherited).
Designator	Abstract type without methods or fields (all declared or inherited, but not from `Object`)
Joiner	Class (or interface) without declared fields or methods that implements an interface (or extends at least two super-interfaces).
Pool	Class or interface without declared instance fields or methods, no visible non-final declared fields and at least one visible static field.
Taxonomy	Class (or interface) that does not declare a method or field and that does not implement an interface (extend a super-interface).
Extender	Class that extends the inherited protocol without overriding visible instance methods.
Implementor	Class of which all declared visible instance methods override an inherited abstract method.
Overrider	Class of which all declared visible instance methods override an inherited concrete method.
Box	Class with a single instance field and at least one mutator method (all declared or inherited).
Canopy	Class with a single instance field and no mutator methods (all declared or inherited).
Compound Box	Class with a single non-primitive instance field and at least one primitive field (all declared or inherited).

Table B.2: Informal μ-pattern descriptions extracted from JTL specifications in [CGM06a].

```
32    jtClassDeclaration(?class,?interp){
33      abstract class ?className *{
34        ![!static ?type ?field = ?init;]
35        ![!abstract !static ?mType::jtType ?mName(?pList) ?sList]
36      }
37    }

38  ?class isPureTypeUnderInterpretation: ?interp if
39    jtClassDeclaration(?class,?interp){
40      abstract class ?className *{
41        abstract ?type::jtType ?methodName(?paramList);
42        ![!abstract ?cType::jtType ?cName(?cPList) ?sList]
43        ![?modList ?fieldType ?field = ?initializer;]
44      }
45    }

46  ?interface isPureTypeUnderInterpretation: ?interp if
47    jtInterfaceDeclaration(?interface,?interp){
48      interface ?interfaceName *{
```

```
49        ?type::jtType ?methodName(?paramList);
50        ![?modList ?fieldType ?field = ?initializer;]
51      }
52    }

53  ?i isStateMachineUnderInterpretation: ?interp if
54    ?i isInterfaceDeclaration,
55    exists(?i interfaceDeclarationDeclaresServiceMethod: ?
56              underInterpretation: ?),
57    forall(?i interfaceDeclarationDeclaresServiceMethod: ?m
58              underInterpretation: ?interp,
59           jtMethodDeclaration(?m,?interp){
60             ?modList ?returnType ?methodName();
61           })

62  ?class isTraitUnderInterpretation: ?interp if
63    jtClassDeclaration(?class,?interp){
64      abstract class ?className *{
65        abstract ?type::jtType ?mName(?paramList);
66        !abstract ?t2::jtType ?m2Name(?p2List) ?statementList
67        ![!static ?t3 ?field = ?init;]
68      }
69    }

70  ?class isRestrictedCreationUnderInterpretation: ?interp if
71    jtClassDeclaration(?class,?interp){
72      class ?className extends* ?super implements* ?interface {
73        ![public ?className(?paramList) ?statementList]
74        static ?type ?field = ?initializer;
75      }
76    },
77    or(?type equals: ?class,
78       ?type equals: ?super,
79       ?type equals: ?interface)

80  ?class isSamplerUnderInterpretation: ?interp if
81    jtClassDeclaration(?class,?interp){
82      class ?className extends* ?super implements* ?interface {
83        public ?className(?paramList) ?statementList
84        static ?type ?field = ?initializer;
85      }
86    },
87    or(?type equals: ?class,
88       ?type equals: ?super,
89       ?type equals: ?interface)

90  ?class isDataManagerUnderInterpretation: ?interp if
91    ?class isClassDeclaration,
92    exists(?class classDeclarationHasServiceMethod: ? underInterpretation: ?),
93    forall(?class classDeclarationHasServiceMethod: ?method
94                  underInterpretation: ?inter,
95           or(?class classDeclarationHasGetterMethod: ?method
96                     forVariableDeclarationFragmentNamed: ?
97                     underInterpretation: ?interp,
98              ?class classDeclarationHasSetterMethod: ?method
99                     forVariableDeclarationFragmentNamed: ?
100                    underInterpretation: ?interp))


101 ?class isRecordUnderInterpretation: ?interp if
102   jtClassDeclaration(?class,?interp){
103     !abstract class ?className *{
```

```
104         public !static ?type ?field = ?init;
105         ![!public !static ?otherType ?otherField = ?otherInit;]
106         ![?modList ?t ?methodName(?paramList) ?statementList]
107       }
108     }

109   ?class isSinkUnderInterpretation: ?interp if
110     jtClassDeclaration(?class,?interp){
111       class ?className {
112         ![?modList ?type ?methodName(?paramList) {
113             ?rec.?inv(?argList);
114           }]
115       }
116     }

117   ?class isCobolLikeUnderInterpretation: ?interp if
118     jtClassDeclaration(?class,?interp){
119       class ?className *{
120         ![!static ?t ?field = ?init;]
121         ![!static ?type::jtType ?methodName(?paramList) ?statementList]
122         ?sMethod := static ?sType::jtType ?sMName(?sPList) ?sSList
123         ![?sMethod :~=  static ?oType::jtType ?oMName(?oPList) ?oSList]
124       }
125     }

126   ?class isFunctionObjectUnderInterpretation: ?interp if
127     jtClassDeclaration(?class,?interp){
128       !abstract class ?className *{
129          ?modList ?t ?field = ?init;
130          ?method := public !static !abstract
131                     ?type::jtType ?mName(?pList) ?sList
132          ![?method :~= public !static !abstract
133                       ?oType::jtType ?omName(?oPList) ?oSList]
134       }
135     }


136   ?class isPoolUnderInterpretation: ?interp if
137     jtClassDeclaration(?class,?interp){
138       class ?className {
139         ![!static ?t ?f = ?i;]
140         ![!static ?tt::jtType ?m(?parameterList) ?statementList]
141         ![!private !final ?ttt ?ff = ?ii;]
142         !private static ?type ?field = ?init;
143       }
144     }

145   ?interface isPoolUnderInterpretation: ?interp if
146     jtInterfaceDeclaration(?interface,?interp){
147       interface ?interfaceName {
148         ![!static ?t ?f = ?i;]
149         ![!static ?tt::jtType ?m(?parameterList);]
150         ![!private !final ?ttt ?ff = ?ii;]
151         !private static ?type ?field = ?init;
152       }
153     }

154   ?interface isTaxonomyUnderInterpretation: ?interp if
155     jtInterfaceDeclaration(?interface,?interp){
156       interface ?emptyInterfaceName extends ?superInterfaceList {
157         ![?modList ?type ?m(?paramList);]
158         ![?modList ?type ?field = ?init;]
```

```
159       }
160     },
161     1 isSizeOf: ?superInterfaceList

162   ?class isTaxonomyUnderInterpretation: ?interp if
163     jtClassDeclaration(?class,?interp){
164       class ?className implements ?interfaceList  {
165         ![?modList ?type ?m(?paramList) ?statementList]
166         ![?modList ?type ?field = ?init;]
167       }
168     },
169     0 isSizeOf: ?interfaceList

170   ?c classDeclarationDeclaresServiceMethod: ?m underInterpretation: ?i if
171     jtClassDeclaration(?c,?i){
172       class ?className  {
173         ?m := [!private !static ?type::jtType ?mName(?pList) ?sList]
174       }
175     }


176   ?class isJoinerUnderInterpretation: ?interp if
177     jtClassDeclaration(?class,?interp){
178       class ?className implements ?interface  {
179         ![?modList ?type ?m(?paramList) ?statementList]
180         ![?modList ?type ?field = ?init;]
181       }
182     },
183     [?interface notNil]

184   ?interface isJoinerUnderInterpretation: ?interp if
185     jtInterfaceDeclaration(?interface,?interp){
186       interface ?emptyInterfaceName extends ?i1, ?i2 {
187         ![?modList ?type ?m(?paramList);]
188         ![?modList ?type ?field = ?init;]
189       }
190     }

191   ?class isCommonStateUnderInterpretation: ?interp if
192     jtClassDeclaration(?class,?interp){
193       class ?className {
194         !final static ?type ?field = ?init;
195       }
196     },
197     jtClassDeclaration(?class,?interp){
198       class ?className *{
199         ![!static ?t ?f = ?i;]
200         ![!static ?methodType::jtType ?methodName(?paramList) ?statementList]
201       }
202     }

203   ?c isImplementorClassUnderInterpretation: ?i if
204     ?c isClassDeclaration,
205     exists(?c classDeclarationDeclaresServiceMethod: ? underInterpretation: ?),
206     forall(?c classDeclarationDeclaresServiceMethod: ?m underInterpretation: ?i,
207            ? abstractMethodIsImplementedBy: ?m inClass: ?c
208              underInterpretation: ?i)


209   ?c isOverriderClassUnderInterpretation: ?i if
210     ?c isClassDeclaration,
211     exists(?c classDeclarationDeclaresServiceMethod: ? underInterpretation: ?),
```

```
212    forall(?c classDeclarationDeclaresServiceMethod: ?m underInterpretation: ?i,
213           ? nonAbstractMethodIsOverriddenBy: ?m inClass: ?c
214             underInterpretation: ?i)


215  ?c isExtenderTypeUnderInterpretation: ?i if
216    ?c isClassDeclaration,
217    exists(?c classDeclarationDeclaresServiceMethod: ? underInterpretation: ?),
218    forall(?c classDeclarationDeclaresServiceMethod: ?m underInterpretation: ?i,
219           ?m methodExtendsProtocolOfClass: ?c underInterpretation: ?i)
```

BIBLIOGRAPHY

[AA01] Hervé Albin-Amiot. JavaXL, a Java source code transformation engine. Technical Report 2001-INFO, École des Mines de Nantes, 2001.

[AACGJ01] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE01)*, pages 166–173, 2001.

[ABL05] Martin Aeschlimann, Dirk Bäumer, and Jerome Lanneluc. Java tool smithing - extending the eclipse java development tools. Presentation at EclipseCON05 available at http://www.eclipsecon.org/2005/tutorials.php, March 2005.

[AF99] Francesca Arcelli and Ferrante Formato. Likelog: a logic programming language for flexible data retrieval. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC99)*, pages 260–267, 1999.

[AK07] Malte Appeltauer and Günter Kniesel. Towards concrete syntax patterns for logic-based transformation rules. In *Proceedings of the Eighth International Workshop on Rule-Based Programming (RULE07)*, 2007.

[AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[All02] Eric Allen. *Bug Patterns in Java*. APress L. P., 2002.

[Als01] Teresa Alsinet. *Logic Programming with Fuzzy Unification and Imprecise Constants: Possibilistic Semantics and Automated Deduction*. Spain, Universitat Politécnica De Catalunya, May 2001.

[Amb] The AmbientTalk project website. http://prog.vub.ac.be/amop/.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education Inc., 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972.

[Bac97] David Francis Bacon. *Fast and effective optimization of statically typed object-oriented languages*. PhD thesis, University of California, 1997.

[BCD+89] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. *SIGPLAN Notices - Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE89)*, 24(2):14–24, 1989.

[BD09] Johan Brichau and Coen De Roover. Language-shifting objects in inter-language interoperability. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST09)*, 2009.

[BDM07] Johan Brichau, Coen De Roover, and Kim Mens. Open unification for program query languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.

[BE03] Godmar Back and Dawson Engler. MJ - a system for constructing bug-finding analyses for Java. Technical report, Stanford University, 2003.

[Bec96] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1996.

[Ber89] J. A. Bergstra. *Algebraic specification*. ACM Press Frontier Series, 1989.

[Bey06] Dirk Beyer. Relational programming with crocopat. In *Proceedings of the 28th International Conference on Software Engineering (ICSE06)*, pages 807–810, 2006.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*. John Wiley & Sons, 2007.

[BNL03] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE03)*, pages 216–225, 2003.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BS02] Patrick Blackburn and Kristina Striegnitz. Natural language processing techniques in Prolog. http://www.coli.uni-saarland.de/ kris/nlp-with-prolog/html/, 2002.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL77)*, pages 238–252, 1977.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) version 1.0. W3C Recommendation, November 1999.

[CEH02] Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in Metal. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE02)*, pages 51–60, 2002.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS86)*, 8(2):244–263, 1986.

[CGM06a] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL and the annoying subtleties of precise μ-pattern definitions. 1st International Workshop on Design Patterns Detection for Reverse Engineering (DPD4RE06/WCRE06), October 2006.

[CGM06b] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL: the Java Tools Language. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA06)*, pages 89–108, 2006.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 1(1):146–166, 1989.

[CH87] Jacques Cohen and Timothy J. Hickey. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS87)*, 9(2):125–163, 1987.

[Che08] Checkstyle project website. http://checkstyle.sourceforge.net/index.html, June 2008.

[CKW93] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[CL96] Yves Caseau and François Laburthe. Claire: Combining objects and rules for problem solving. In *Proceedings of the JICSLP workshop on Multi-Paradigm Logic Programming (MPLP96)*, pages 105–114, 1996.

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering (ICSE92)*, pages 138–156, 1992.

[Cop91] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Publishing Company, September 1991.

[Cre97] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages (DSL'97)*, pages 229–242, 1997.

[CW96] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM). ACM*, 43(1):20–74, 1996.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles (SOSP01)*, pages 73–88, 2001.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99)*, pages 411–420, 1999. http://patterns.projects.cis.ksu.edu/.

[DBD06] Coen De Roover, Johan Brichau, and Theo D'Hondt. Combining fuzzy logic and behavioral similarity for non-strict program validation. In *Proceedings of the 8th ACM-SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP06)*, pages 15–26, 2006.

[DBN+07] Coen De Roover, Johan Brichau, Carlos Noguera, Theo D'Hondt, and Laurence Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM07)*, pages 92–102, 2007.

[DdMS02] Stephen Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 133–144, 2002.

[De 04] Coen De Roover. Incorporating dynamic analysis and approximate reasoning in declarative meta-programming to support software reengineering. Master's thesis, Vrije Universiteit Brussel, 2004.

[De 06] Kris De Volder. JQuery: A generic code browser with a declarative configuration language. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL06)*, pages 88–102, 2006.

[Dev92] Premkumar T. Devanbu. GENOA: a customizable language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software engineering (ICSE92)*, pages 307–317, 1992.

[Dev99] Premkumar T. Devanbu. GENOA–a customizable, front-end-retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 8(2):177–212, 1999.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP95)*, pages 77–101, 1995.

[DGD05] Coen De Roover, Kris Gybels, and Theo D'Hondt. Towards abstract interpretation for recovering design information. In *Proceedings of*

the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL05), volume 131 of *Electronic Notes in Theoretical Computer Science*, pages 15–25, May 2005.

[DGJ04] Maja D'Hondt, Kris Gybels, and Viviane Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 2004 ACM symposium on Applied computing (SAC04)*, pages 1328–1335, 2004.

[DHK96] Arie Van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., 1996.

[dLW03] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-order and Symbolic Computation*, 16(1-2):15–35, 2003.

[DMG+06] Coen De Roover, Isabel Michiels, Kim Gybels, Kris Gybels, and Theo D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC06)*, pages 202–211, 2006.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI00)*, October 2000.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP01)*, pages 57–72, 2001.

[Ecl08a] The Eclipse JDT Core Component Website. http://www.eclipse.org/jdt/core/index.php, 2008.

[Ecl08b] Eclipse website. http://www.eclipse.org/, June 2008.

[EK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, October 1976.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA)*, pages 24–27, May 2003.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[Fis05] Dale Fisk. Programming with punched cards. http://www.columbia.edu/acis/history/fisk.pdf, 2005.

[FKI+07] Henry Falconer, Paul H. J. Kelly, David M. Ingram, Michael R. Mellor, Tony Field, and Olav Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th International Conference on Compiler Construction (CC07)*, 2007.

[FM04] Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. *Elsevier International Journal on Computer Languages, Systems & Structures*, 30(1-2):21–33, 2004.

[FNT+98] Thorsten Fischer, Jörg Niere, Lars Torunski, , and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In *Proceedings of the 6th International Workshop on Theory and Applications of Graph Transformation (TAGT98)*, number 1764 in Lecture Notes in Computer Science, 1998.

[Fow97] Martin Fowler. *UML Distilled*. Addison Wesley, 1997.

[GAA01] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the 39th Conference on the Technology of Object-Oriented Languages and Systems (TOOLSUSA01)*, pages 296–305, 2001.

[GAM96] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the 4th International Workshop on Program Comprehension (IWPC96)*, page 144, 1996.

[GC08] Yaser Ghanam and Sheelagh Carpendale. A survey paper on software architecture visualization. Technical Report 2008-906-19, Department of Computer Science, University of Calgary, Canada T2N 1N4, 2008.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GJ90] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide*. Ellis Horwood, 1990.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[GM05] Joseph (Yossi) Gil and Itay Maman. Micro patterns in java code. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA05)*, pages 97–116, 2005.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[GS00] David Gilbert and Michael Schroeder. Fury: Fuzzy unification and resolution based on edit distance. In Nikolaos G. Bourbakis, editor, *Proceedings of the 1st IEEE International Symposium on Bioinformatics and Biomedical Engineering (BIBE00)*, pages 330–336, November 2000.

[Gué03] Yann-Gaël Guéhéneuc. *Un cadre pour la tracabilite des motifs de conception.* PhD thesis, Ecole des Mines de Nantes, June 2003.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Elesevier International Journal on Computer Languages, Systems and Structures*, 32:109–124, 2006.

[Háj98] Petr Hájek. Deductive systems of fuzzy logic (a tutorial). Tutorial, 1998.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI02)*, pages 69–82, 2002.

[Hea78] H. S. Heaps. *Information Retrieval.* Academic Press, 1978.

[HGC+07] Charlotte Herzeel, Kris Gybels, Pascal Costanza, Coen De Roover, and Theo D'Hondt. Forward chaining in HALO: An implementation strategy for history-based logic pointcuts. In *Proceedings of the 2007 international conference on Dynamic languages (ICDL07)*, pages 157–182, 2007.

[HGC+09] Charlotte Herzeel, Kris Gybels, Pascal Costanza, Coen De Roover, and Theo D'Hondt. Forward chaining in HALO: An implementation strategy for history-based logic pointcuts. *Elsevier International Journal on Computer Languages, Systems & Structures*, 35(1):31–47, April 2009.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE01)*, pages 54–61, 2001.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA02)*, pages 161–173, 2002.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[HSD08] Jeffrey S. Hammond, Carey Schwaber, and David D'Silva. IDE usage trends - Forrester Research. http://www.microsoft.com/presspass/itanalyst/docs/02-12-08IDEUsageTrendsJeffreyHammond.PDF, February 2008.

[HVd06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, 2006.

[Jav] JavaConnect project website. http://www.info.ucl.ac.be/ jbrichau/software.html.

[JD03] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD03)*, pages 178–187, 2003.

[jHo07] jHotDraw project website. http://www.jhotdraw.org/, 2007.

[Joh79] Stephen C. Johnson. *YACC: Yet Another Compiler-Compiler*. Unix Programmer's Manual Vol 2b, 1979.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA92)*, pages 63–72, 1992.

[KBD08] Andy Kellens, Johan Brichau, and Coen De Roover. Example-based program querying. In *Proceedings of the 2008 Working Session on Query Technologies and Applications for Program Comprehension (QTAPC08) at the 16th International Conference on Program Comprehension (ICPC08)*, 2008.

[KC99] E. E. Kerre and M. De Cock. Linguistic modifiers: An overview. *Fuzzy Logic and Soft Computing*, pages 69–85, 1999.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, 1996.

[Kli03] Paul Klint. How understanding and restructuring differ from compiling - a rewriting perspective. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 03)*, page 2, 2003.

[Kli05] Paul Klint. A tutorial introduction to rscript — a relational approach to software analysis, draft. Technical report, Centrum voor Wiskunde en Informatica, May 2005.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[KP96] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 208, 1996.

[KPK94] George Kiczales, Andreas Paepcke, and Gregor Kiczales. *Open Implementations and Metaobject Protocols*. MIT Press, 1994.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st International Conference on Software engineering*, pages 226–235, 1999.

[Lac03] David Lacey. *Program Transformation using Temporal Logic Specifications*. PhD thesis, University of Oxford, August 2003.

[LBH+08] Julia L. Lawall, Julien Brunel, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. Technical Report 08/1/INFO, Ecole des Mines de Nantes, 2008.

[LdM01] David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *Proceedings of the 10th International Conference on Compiler Construction (CC01)*, pages 52–68, 2001.

[Lee72] Richard C. T. Lee. Fuzzy logic and the resolution principle. *Journal of the ACM*, 19(1):109–119, 1972.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Proceedings of the 12th International Conference on Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169, April 2003.

[LH06] Ondrej Lhoták and Laurie J. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proceedings of the 15th International Conference on Compiler Construction (CC06)*, Lecture Notes in Computer Science, pages 47–64, 2006.

[Lho02] Ondrej Lhoták. Spark: A flexible points-to analysis framework for java. Master's thesis, McGill University, December 2002.

[Lho06] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[Liv05] Benjamin Livshits. Turning Eclipse against itself: Finding bugs in Eclipse code using lightweight static analysis. In *EclipseCON05 Research Exchange*, March 2005.

[Liv06] Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, 2006.

[LL90] Deyi Li and Dongbo Liu. *A fuzzy Prolog database system*. John Wiley & Sons, Inc., 1990.

[LL07] Jonas Lundberg and Welf Löwe. A scalable flow-sensitive points-to analysis. In *Advances and Applications in Compiler Construction, Festschrift on the occasion of the retirement of Prof. Dr. Dr. h.c. Gerhard Goos*, 2007.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, 2006.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.

[LR94] Ladd and Ramming. A*: a language for implementing language processors. *Proceedings of the 1994 International Conference on Computer Languages (ICCL94)*, pages 1–10, 1994.

[LR01] Ralf Lämmel and Günter Riedewald. Prological language processing. In *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA01)*, volume 44, 2001.

[LRY+04] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI04)*, pages 219–230, 2004.

[LWL+05a] Monica Lam, John Whaley, Benjamin Livshits, Michael Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS05)*, pages 1–12, 2005.

[LWL05b] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. Technical report, Stanford University, 2005.

[MDB+06] Isabel Michiels, Coen De Roover, Johan Brichau, Elisa Gonzalez Boix, and Theo D'Hondt. Program testing using high-level property-driven models. In *Proceedings of the Eighteenth International Conference on Software Engineering and Knowledge Engineering (SEKE06)*, pages 489–494, 2006.

[Mer03] Jason Merrill. GENERIC and GIMPLE: a new tree representation for entire functions. In *Proceedings of the 2003 GCC & GNU Toolchain Developers' Summit (GCC03)*, pages 171–180, 2003.

[Mic99] Sun Microsystems. *Code Conventions for the Java Programming Language*. Sun Microsystems, Inc., 1999.

[MLL05] Michael Martin, Benjamin Livshits, and Monica Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA05)*, pages 365–383, 2005.

[MOAV04] Jesús Medina, Manuel Ojeda-Aciego, and Peter Vojtás. Similarity-based unification: A multi-adjoint approach. *Fuzzy Sets and Systems - Selected Papers from EUSFLAT 2001*, 146(1), August 2004.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE01)*, page 13, 2001.

[Mos05] Maxim Mossienko. Structural search and replace: What, why, and how-to. JetBrains onBoard Online Magazine, February 2005. `http://www.onboard.jetbrains.com/is1/articles/04/10/ssr/`.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA02)*, pages 1–11, 2002.

[Nai96] Lee Naish. Higher-order logic programming in Prolog. Technical Report 96/2, University of Melbourne, Australia, 1996.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005. Corrected 2nd printing.

[NSW+02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE02)*, pages 338–348, 2002.

[NWW03] Jörg Niere, Jörg P. Wadsack, and Lothar Wendehals. Handling large search space in pattern-based reverse engineering. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC03)*, page 274, 2003.

[OO90] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, 1990.

[Opd92] William F. Opdyk. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Our89] Dirk Ourston. Program recognition. *IEEE Expert: Intelligent Systems and Their Applications*, 4(4):36–49, 1989.

[Pau92] Santanu Paul. SCRUPLE: a reengineer's tool for source code search. In *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative research (CASCON92)*, pages 329–346, 1992.

[Pee87] Howard A. Peell. An APL idiom inventory. *ACM SIGAPL APL Quote Quad*, 17(4):362–368, 1987.

[PK98] Lutz Prechelt and Christian Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *The Journal of Universal Computer Science*, 4(11):866–882, 1998.

[PLM06] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers. *Electronic Notes in Theoretical Computer Science - Proceedings of the 2006 International ERCIM Workshop on Software Evolution*, 166:47–62, 2006.

[PMD08] PMD project website. http://pmd.sourceforge.net/, June 2008.

[PNP06] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA Futurs, Projet Jacquard, LIFL, Lille, France, May 2006.

[PP94] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[PS02] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. Microtome Publishing, 2002. Millenial reissue of original 1987 print.

[PULPT02] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, 28(6):595–606, 2002.

[PW80] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[PW83] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of 21st Annual Meeting of the Association for Computational Linguistics (AMACL83)*, pages 137–144, 1983.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[RC97] R. Ramesh and Weidong Chen. Implementation of tabled evaluation with delaying in Prolog. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):559–574, 1997.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM99)*, pages 13–22, 1999.

[Ric02] Tamar Richner. *Recovering Behavioral Design Views: a Query Based Approach*. PhD thesis, Universität Bern, Philosophisch-naturwissenschaftlichen Fakultät, May 2002.

[Rie00] Dirk Riehle. *Framework Design: A Role Modeling Approach.* PhD thesis, ETH Zürich - Institute of Computer Systems, 2000.

[Riv96] Fred Rivard. Smalltalk: a reflective language. *Reflection*, 1996.

[RKA06] Tobias Rho, Günter Kniesel, and Malte Appeltauer. Fine-grained generic aspects. In *Proceedings of the AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL06)*, 2006.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA01)*, volume 36 of *SIGPLAN Notices*, pages 43–55, 2001.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV97)*, pages 143–154, 1997.

[RW90] Charles Rich and Linda Mary Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 07(1):82–89, 1990.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Görel Hedin, editor, *Proceedings of the 12th International Conference on Compiler Construction (CC2003)*, volume 2622, pages 126–137, April 2003.

[RZ96] Dirk Riehle and Heinz Züllighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems*, 2(1):3–13, 1996.

[SB05] Volker Stolz and Eric Bodden. Temporal assertions using AspectJ. In *Proceedings of the Fifth International Workshop on Run-time Verification (RV05)*, 2005.

[SDF08] SDF: Language for modular syntax definition. http://www.program-transformation.org/Sdf/WebHome, June 2008.

[Ses02] Maria I. Sessa. Approximate reasoning by similarity-based sld resolution. *Theoretical Computer Science*, 275(1-2):389–426, 2002.

[Shi04] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. *ACM SIGPLAN Notices*, 39(4):257–269, 2004.

[Sou08] The Smalltalk Open Unification Language (SOUL). http://prog.vub.ac.be/SOUL/, 2008.

[SR05] Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In Pedro Barahona and Amy P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP05)*, pages 117–128, July 2005.

[Sri07] Manu Sridharan. *Refinement-Based Program Analysis Tools*. PhD thesis, EECS Department, University of California, Berkeley, October 2007.

[SS06] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 2006 ASPLOS Conference (ASPLOS'06)*, pages 416–425, 2006.

[Str06] Umberto Straccia. A simple top-down query answering procedure for many-valued logic programming. Technical report, ISTI - CNR, April 2006.

[SV98] Alex Sellink and Chris Verhoef. Native patterns. In *Proceedings of the Working Conference on Reverse Engineering (WCRE98)*, page 89, 1998.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of the International Symposium on Principles of Software Evolution (*, pages 154–164, 2000.

[Tor96] Olof Torgersson. A note on declarative programming paradigms and the future of definitional programming. Proceedings of Das Winteröte '96, 1996.

[vdBHKO02] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[vdBKV07] Mark van den Brand, Paul Klint, and Jurgen Vinju. The language specification formalism ASF+SDF. http://homepages.cwi.nl/ daybuild/daily-books/extraction-transformation/asfsdf/asfsdf.html, 10 2007.

[vE86] M H van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 30(1):37–53, 1986.

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: a scripting language for refactoring. In Dieter Rombach and Mary Lou Soffa, editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE06)*, pages 172–181, 2006.

[VGMH02] Claudio Vaucheret, Sergio Guadarrama, and Susana Muñoz-Hernández. Fuzzy Prolog: A simple general implementation using CLP(R). In Peter J. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming (ICLP02)*, volume 2401 of *Lecture Notes in Computer Science*, page 469, 2002.

[vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[Vol06a] Nic Volanschi. Condate: a proto-language at the confluence between checking and compiling. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP06)*, pages 225–236, 2006.

[Vol06b] Nic Volanschi. A portable compiler-integrated approach to permanent checking. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE06)*, pages 103–112, 2006.

[VR08] Nic Volanschi and Christian Rinderknecht. Unparsed patterns: easy user-extensibility of program manipulation tools. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM08)*, pages 111–121, 2008.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON99)*, page 13, 1999.

[VS00] Joost Visser and Jeroen Scheerder. A quick introduction to SDF. ftp://ftp.stratego-language.org/pub/stratego/docs/sdfintro.pdf, April 2000.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118, 2005.

[War83] David H. D. Warren. An abstract Prolog instruction setrolog instruction set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, October 1983.

[Wha03] John Whaley. Joeq: a virtual machine and compiler infrastructure. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME03)*, pages 58–66, 2003.

[Wil92] Linda Mary Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, Massachusetts Institute of Technology, July 1992.

[Wil93] Linda Mary Wills. Flexible control for program recognition. In *Proceedings of the Working Conference on Reverse Engineering (WCRE93)*, pages 134–143, 1993.

[Wil94] Linda Mary Wills. Using attributed flow graph parsing to recognize clichés in programs. In *Proceedings of the 5th International Workshop on Graph Grammars and Their Application to Computer Science*

(TAGT94), volume 1073 of *Lecture Notes in Computer Science*, pages 170–184, 1994.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI04)*, pages 131–144, 2004.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the 26th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS98)*, pages 112–124, 1998.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, Belgium, January 2001.

[XSB07] The XSB system version 3.1 volume 1: Programmer's manual. http://xsb.sourceforge.net/manual1/index.html, 2007.

[Zad65] Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.