Declarative Meta Programming to Support Software Development

Workshop Proceedings

Tom Mens
Programming Technology Lab
Vrije Universiteit Brussel, Belgium
tom.mens@vub.ac.be

Roel Wuyts
Software Composition Group
University of Bern, Switzerland
roel.wuyts@iam.unibe.ch

Kris De Volder
Department of Computer Science
University of British Columbia, Canada
kdvolder@cs.ubc.c

Kim Mens
Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
kim.mens@info.ucl.ac.be

Table of Contents

Declarative Meta Programming to Support Software Development: Workshop Report

Tom Mens*
Programming Technology Lab
Vrije Universiteit Brussel, Belgium
tom.mens@vub.ac.be

Roel Wuyts
Software Composition Group
University of Bern, Switzerland
roel.wuyts@iam.unibe.ch

Kris De Volder
Department of Computer Science
University of British Columbia, Canada
kdvolder@cs.ubc.ca

Kim Mens
Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
kim.mens@info.ucl.ac.be

Abstract

This paper reports on the results of the workshop on *Declarative Meta Programming to Support Software Development* in Edinburgh on September 23, 2002. It enumerates the presentations made, classifies the contributions and lists the main results of the discussions held at the workshop. As such it provides the context for future workshops around this topic.

Keywords: meta programming, declarative languages, software development

Introduction

The workshop on *Declarative Meta Programming to Support Software Development* (DMP 02) was co-located with the *17th International Conference on Automated Software Engineering* (ASE 2002), and took place at the Heriot-Watt University in Edinburgh, United Kingdom, on September 23, 2002. There were 13 participants, most of which contributed with a position paper that was reviewed and revised before the workshop. The participants originated from Belgium, Canada, France, Switzerland, Israel, United Kingdom, and the USA.

The workshop focused on declarative meta programming (DMP) techniques and tools to support software development. Such techniques and tools are *meta programming* because they reason about or manipulate program code at a meta level to automate some aspect of the software development process. The fact that they are *declarative* means that they focus on *what* is being done rather than *how* it is done.

The workshop had the following explicit goals:

- Get an overview of existing DMP approaches.
- Delineate for which software development activities DMP could be used.
- Compare existing approaches (tools, techniques and formalisms) and identify commonalities and differences.
- Discuss advantages and shortcomings of DMP for supporting software development.

*Tom Mens is a postdoctoral fellow of the Fund for Scientific Research - Flanders (Belgium).

Workshop presentations

The morning session was devoted to four long presentations of 20 minutes and four short presentations of 10 minutes, each followed by 5 minutes of discussion. The long presentations were chosen by the organisers because they offered different or novel perspectives on the workshop topic, and because they had a higher potential for generating issues that would stimulate the discussions.

The papers and their authors were as follows, with the names of the actual presenters during the workshop underlined. The papers were collected in a technical report [WMDM02].

Long presentations:

LP1 Toacy Oliveira, Paulo Alencar, Donald Cowan (University of Waterloo, Canada). Towards a declarative approach to framework instantiation.

LP2 Yann-Gaël Guéhéneuc (École des Mines de Nantes, France). Meta-modelling, logic programming, and explanation-based constraint programming for pattern description and detection.

LP3 Tom Tourwe, Johan Brichau, Tom Mens (Vrije Universiteit Brussel, Belgium). Using declarative metaprogramming to detect possible refactorings.

LP4 Gopal Gupta (University of Texas, Dallas, USA). A language-centric approach to software engineering: domain specific languages meet software components.

Short presentations:

SP1 Tom Tourwé, Tom Mens (Vrije Universiteit Brussel, Belgium). A declarative meta-programming approach to framework documentation.

SP2 H. Akehurst, Behzad Bordbar, P.J.Rodgers, N.T.G. Dalgliesh (University of Kent, United Kingdom). Automatic normalisation via metamodeling.

SP3 Robert Filman, Klaus Havelund (NASA Ames Research Center, California, USA). Realising aspects by transforming for events.

SP4 Greg Michaelson (Heriot-Watt University, United Kingdom). SML prototypes from Z specifications.

SP5 Cordell Green (Kestrel Institute, USA). SpecWare: Automatic formal specifications into hardware.

According to the workshop topic, the papers could be classified according to two dimensions: the kind of DMP technique they use (see Table 1) and the kind of support for software development they provide (see Table 2).

Presentation	DMP approach used
LP1	annotated UML, XML, XSLT
LP2	meta modelling, logic programming, explanation-based constraint programming
LP3	logic meta programming
LP4	constraint logic programming, denotational semantics
SP1	logic meta programming
SP2	OCL, graph rewriting
SP3	declarative language
SP4	translation scheme
SP5	theorem provers

Table 1: Declarative Meta Programming approach used

In Table 1 we observe that most of the presented declarative meta programming approaches use some variant of logic meta programming (LP2, LP3, LP4, SP1). Other approaches use more trendy languages and standard technologies such as UML, OCL, XMI, XML and XSLT (LP1, SP2).

Presentation	Kind of development support
LP1	framework documentation
LP2	design patterns
LP3	design patterns, refactoring
LP4	domain-specific languages
SP1	framework instantiation and evolution
SP2	database normalisation
SP3	aspect-oriented programming
SP4	program translation
SP5	code generation from formal specifications

Table 2: Kind of software development support

As can be seen from Table 2, the bulk of the presented approaches uses declarative meta programming to provide support for developing *object-oriented* software applications (LP1, LP2, LP3, SP1, SP3). This support includes: documentation, instantiation and evolution of object-oriented application frameworks; description, detection, generation and conformance checking of design pattern instances; object-oriented refactoring; aspect-oriented programming.

Workshop discussions

In order to stimulate discussions, some general important questions were posed to the participants during the workshop:

Q1 What are the main advantages of DMP over other approaches?

The following benefits were mentioned by the participants:

- *Portability* and *platform independence*. For example, if we express domain-specific languages with DMP, they can be automatically translated to any target platform.
- Declarative programs provide an *executable form of documentation*. *Executable*, since they are programs; *documentation*, since the declarative notation is easy to read and understand.
- *Conciseness* and *complexity reduction*. Declarative programs are often significantly smaller and less complex than non-declarative programs. Cordell Green mentioned an experimentally validated factor 2 to 5 reduction of program dependencies.
- *Error reduction*. This is a direct consequence of complexity reduction. Cordell Green cited an experimentally validated error reduction of a factor 2 to 20.

Q2 What are the potential shortcomings of DMP?

- *Performance* and *efficiency* issues were coined as a potential disadvantage of DMP, but most of the participants agreed that this was a non-issue. With the current state-of-the-art in compiler technology, very efficient logic languages can be implemented.
- Declarative meta programming involves a *high degree of sophistication*. It requires a deep understanding of language semantics. This is even more the case with hybrid DMP, for example when a declarative meta language is used on top of an object-oriented base language. In that case, complex issues such as language symbiosis come into play. As a result, DMP is not suited for the average programmer, and it will never find widespread use. This resulted in the third question to be discussed:

Q3 How can DMP achieve more widespread acceptance as a mechanism for supporting software development?

- *Lack of standard technologies* was suggested as a reason why declarative languages have not found wide adoption for software development support. This can be resolved relatively easy by putting an XML-syntax on top of the declarative language, at the expense of losing the more concise and readable notation.
- A second aspect that strongly affect acceptance of DMP is the *quality and usability of the supported tools*. Two powerful and promising tools for DMP were presented at the end of the day, and are discussed later in this paper.

Q4 For which kinds of support for software development is DMP well-suited/unsuited?

This final question was only discussed very briefly due to time constraints. *Parse tree manipulation* was proposed as

something for which DMP is particularly well suited. Indeed, many of the presented approaches used or proposed some kind of parse tree manipulation for generating, transforming or reasoning about code.

Tool demonstrations

Upon explicit request by the workshop participants, a special tool demonstration session was scheduled at the end of the day, where two sophisticated DMP tools for reasoning about object-oriented programs (one for Smalltalk and one for Java) were demonstrated.

The first tool, *Soul* [MMW02] was presented by Johan Brichau. It is a Prolog-like logic meta programming language built on top of, and tightly integrated with, a Smalltalk object-oriented software development environment. It enables support for design patterns, coding conventions, programming styles, refactoring, and software metrics.

The second demonstration was made by Yann-Gaël Guéhéneuc and showed the *Patternsbox* tool (that allows to select and instantiate patterns) and the *PtiDej* tool (that does program architecture visualization and patterns detection). These tools allow to specify (patterns), and then use these specifications to generate code or check the specification against Java source code. One of the very nice features is that it employs a constraint system that gives feedback on how well the patterns match the code. Hence the pattern serves more as a fuzzy definition that can yield partial matches, and it explains these results.

Acknowledgements

This workshop was supported by the *Scientific Research Network on Foundations of Software Evolution* [ESF02].

References

[ESF02] Fund for Scientific Research - Flanders (Belgium). *Scientific Research Network on Foundations of Software Evolution.*
http://prog.vub.ac.be/FFSE [1 Oct 2002]

[GDJ02] Yann-Gaël Guéhéneuc, Rémi Douence and Narendra Jussien. No Java without Caffeine: A tool for dynamic analysis of Java programs. In *Proc. Int'l Conf. Automated Software Engineering*, pages 117-126, Edinburgh, United Kingdom, September 2002. IEEE Computer Society Press.

[MMW02] Kim Mens, Isabel Michiels, Roel Wuyts. Supporting Software Development through Declaratively Codified Programming Patterns. *Journal on Expert Systems with Applications*, December 2002. Elsevier Publications.

[WMDM02] Roel Wuyts, Tom Mens, Kris De Volder and Kim Mens. Proc. of the *Workshop on Declarative Meta-Programming to Support Software Development*. Technical Report VUB-PROG-TR-??-2002, Programming Technology Lab, Vrije Universiteit Brussel, 2002.
http://www.cs.ubc.ca/ kdvolder/Workshops/ASE2002/DMP/ [1 Oct 2002]

Towards a Declarative Approach to Framework Instantiation

Toacy C. Oliveira♦, Paulo S. C. Alencar*, Donald D. Cowan*
Computer Science Department, University of Waterloo – Waterloo, Canada
{toliveira, palencar, dcowan@csg.uwaterloo.ca}

Abstract

Object-oriented frameworks are currently regarded as a promising technology for reusing designs and implementations. However, developers find there is still a steep learning curve when extracting the framework design rationale and understanding the documentation during framework instantiation. Thus, instantiation is a costly process in terms of time, people and other resources. Problems like: obtaining design rationale through "Code Mining"; understanding the instantiation process commonly described in natural language; and violating "good" design principles and/or domain constraints, frequently emerge throughout the framework's reuse process. In this paper we present a process-based approach to framework instantiation that addresses these issues. Our main goal is to represent the framework architectural design models in an explicit and declarative way, and support changes to this architecture based on explicit instantiation processes and activities while maintaining system integrity, invariants, and general constraints.

1 Introduction

Current resource and time-to-market constraints on the software development process push developers to create a final product as quickly as possible. To accomplish such a "mission" developers need to base their development on successful past experiences [1]. Moreover, they should re-use these experiences in well-organized computer-aided processes that avoid and/or highlight any violations of functional and non-functional constraints. In this context, Object Oriented Frameworks [2][3] appear to be a promising approach if 1) the developers can rely on the design and implementation of a semi-complete application; 2) frameworks can be decorated with design constraints to aid integrity checking and; 3) frameworks can be embedded in a development environment.

In order to obtain a complete application, framework reusers [4] must follow an instantiation process. In this process, a framework is completed by the addition of new design elements that can be viewed as application-specific increments that produce the final product [5] (Figure 1).

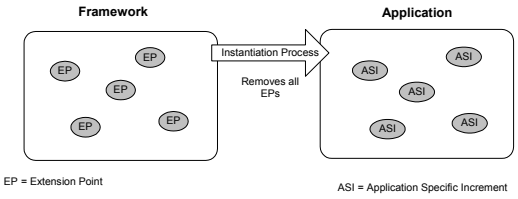

Figure 1 – Framework and it's instantiation process.

It is important to notice that the addition of such increments can: 1) be a lengthy and complicated process with numerous conditional instantiation rules; 2) corrupt the framework meta-level constraints in terms of hierarchy depth, number of methods and coupling, thus violating design principles; 3) cause bugs or errors in the final application.

To mitigate some of these problems we are investigating a process-based approach to framework instantiation where: 1) both framework and final application designs are expressed in XMI [6]; 2) the processes are specified in XML and; 3) the instantiation execution is based on the declarative language XSLT; 4) meta-level constraints are expressed in XSLT.

Using this approach we are able to: 1) represent the instantiation process design in a computer "manipulable" manner; 2) manipulate the framework meta-level using a declarative approach; 3) be compatible with major CASE tools and development environments once we use XMI, XML and XSLT as standard forms of representation; 4) analyse the final design meta-level constraints using a declarative approach; 5)

♦Sponsored by CNPq – Brazil and *NSERC - Canada

highlight design elements that are suitable for reengineering based on design analysis;6) replicate the instantiation process itself.

The remainder of this paper contains a description of the approach and indicates future directions for research.

2 Approach

Framework instantiation is the ability to combine a previous semi-complete object-oriented architecture with ASIs[5] (Application Specific Increments) to meet the needs of the current application. In order to achieve this goal, software developers practicing re-use (reusers) usually have to read unstructured documentation, with unpredictable and untraceable consequences. To alleviate such problem we have developed a set of well-structured documents that are introduced in a computer-based environment to guide reusers throughout the process, in a step-by-step procedure.

In figure 2, we indicate the notion of Reusable Artefact, which is a set of structured documents required for framework instantiation. These documents[1] are: 1) the Framework Design Representation, expressed in UML-FI (UML – Framework Instantiation) that identifies the placeholders for the framework extension at the design level; 2) the Instantiation Specification Cookbook, expressed in RDL (Reuse Description Language) that contains the step-by-step representation of the instantiation process; 3) a set of design principles expressed in XSLT.

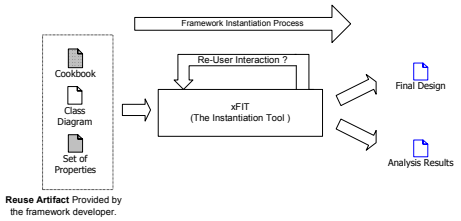

Figure 2 – The Approach overview.

During the instantiation process, the reusable artefact is provided as input to xFIT (XML based Framework Instantiation Tool), an engine that interacts with the reuser to capture the ASI semantic in a window-based environment. This ASI semantic is then used in the Framework Design in order to 'fill its empty spaces', resulting in a complete final design.

Once the instantiation process is complete, non-functional and functional constraints can be evaluated in order to check system/design integrity.

2.1 Extension Point Representation – UML-FI

Extension points are placeholders that indicate where a framework can be adapted. Our approach focuses on instantiation of object-oriented frameworks, and so we need to be able to express the nature of an extension point and the related instantiation activities in terms of OO programming techniques. This representation uses a UML Decorated Class Diagram, providing reusers with an overview at the design level of the OO activities that should be performed to obtain reuse. Moreover, it provides an accurate representation of the effort needed to obtain a valid instance and a way to classify frameworks.

We use the term object-oriented extensions points (OOEP for short) to designate the elements in an OO design that are suitable for instantiation. At the present, OOEP can be related to: classes, methods or attributes. We also use the term instantiation task (IT for short) as the task that should be executed to instantiate/complete an OOEP. Table 1 contains the types of OOEP and their associated IT.

Table 1 – OOEP and Associated IT

OOEP	IT
Class	**Class Extension** – Creates a subclass of a given class. **Select Class Extension** – Chooses a subclass of a given class among its sub-classes. **Pattern Class Extension** - Creates a subclass of a given class through pattern application.
Method	**Method Extension** - Creates a method in a subclass that is a redefinition of the super class. **Pattern Method Extension** - Creates a method in a subclass through pattern application.
Attribute	**Value Assignment** – Assigns a value to an attribute. **Value Selection** - Assigns a value to an attribute from a list of values.

[1] At the time this paper was written, domain properties had not yet been introduced.

Examining Table 1 we notice that we have adopted the concept of "Pattern Instantiation". A Pattern Instantiation means that the associated instantiation will be driven by a pattern of correlated actions. This mechanism allows reuse of well-known design structures (micro-architectures). Design Patterns [7] are examples of this approach.

We've also introduced the concept that elements in the framework design can be mandatory or optional. A mandatory element is an essential element to the framework execution and should be present in the final design. On the other hand, optional elements represent an add-on to the framework's core functionality and can be omitted if necessary. This extra concept introduces a new IT that reflects **selection** of a design element.

The incorporation of all ITs in UML (deriving UML-FI) is done through stereotype and tagged values mechanisms. This approach facilitates the evaluation of the instantiation effort by visual inspection of a class diagram (Figure 3).

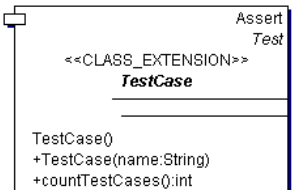

Figure 3 –Class Extension IT visualization.

2.2 Instantiation Specification

The representation provided by UML-FI class diagrams cannot by itself be a complete guide to a valid framework instance. Some information such as what pattern to apply, was deliberately 'forgotten' to avoid graphical complexity. To accommodate the missing information such as sequencing and dependency, we are currently transforming our previous procedural based approach [8][9] to a declarative XSLT base. As in RDL (Reuse Description Language), the declarative approach should be able to specify the manipulation of the framework meta-level to incorporate new design elements as necessary. For example, if the framework developer specifies that a class should be extended, the XSLT should introduce a new class into the final application design (Figure 4).

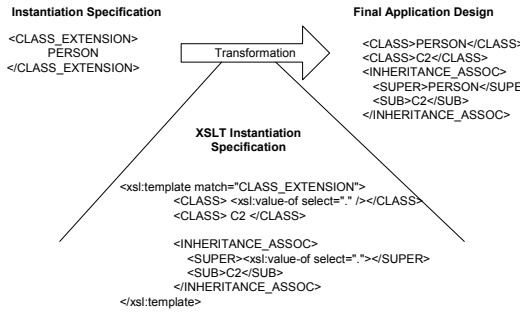

Figure 4 – Instantiation Specification

It is important to notice that after the transformation we still need reuser participation. For example, the reuser could provide meaningful names to classes (in opposition to C2 as can be seen in figure 4).

Such a declarative approach proved to be extensible to all RDL specifications that involve design meta-level manipulation. We are now investigating how to introduce RDL sequencing and conditional aspects.

2.3 Analysis

The analysis phase of our approach is divided in two categories: Structural Analysis and Behavioral Analysis. Structural Analysis examines whether the documents involved in the process are "well-formed." For example, the final application is characterized by the absence of OOEP. Thus, the application's final design should be checked to confirm the presence or absence of such features. In the same way, the application can be audited for "good" design principles to avoid or emphasize future reengineering needs.

Behavioral Analysis is about aspects of the execution of the instantiation process and properties of the final application such as liveness, reachability and deadlocks. For example, the instantiation processes are by nature asynchronous [10]. This means that a deadlock situation may arise.

In our approach Structural Analysis is achieved using a declarative approach (XSLT) that checks the desired feature (Figure 5).

```
<xsl:template match="XMI.content">
    <xsl:for-each select="Foundation.Extension_Mechanisms.Stereotype">
      <xsl:if test="contains(Foundation.Core.ModelElement.name,'CLASS_EXTENSION')">
            <CLASS_TAG>
                  <xsl:value-of select="Foundation.Core.ModelElement.name"> </xsl:value-of>
            </CLASS_TAG>
       </xsl:if>
       <xsl:if test="contains(Foundation.Core.ModelElement.name,'METHOD_EXTENSION')">
            <MET_TAG>
                  <xsl:value-of select="Foundation.Core.ModelElement.name"> </xsl:value-of>
            </MET_TAG>
      </xsl:if>
    </xsl:for-each>
</xsl:template>
```

Figure 5 – XSLT for detecting the presence of EP in the final design.

Behavior will probably be described with a declarative approach using XL (a process language). Behavioral analysis will be achieved by translating the XL description into μ-calculus and then verifying the behavioural properties through model checking techniques.

3 Conclusions & Future Works

Object-oriented frameworks can demand lengthy and complicated instantiation processes. Therefore, the instantiation process needs to be supported by computer-based tools in order to reduce or even eliminate mistakes. In this paper we have presented an approach to framework instantiation that combines UML and XML standards with the expressiveness of declarative design manipulation to handle such a process.

Our approach differs from the previous ones [11][12][13][14], as we are able to specify instantiation process characteristics like, sequencing and dependency through unambiguous representations. In addition, as in [15], the declarative approach facilitates design meta-level manipulation/analysis, enabling for example, automatic Pattern application/recognition.

The next steps we expect to attempt are; 1) develop a domain constraint analysis to allow domain specific reasoning; 2) integrate a requirements specification approach to allow framework reuse from an even higher level viewpoint.

4 Bibliography

[1]Jacobson, I.; Griss, M.; Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, Reading, Massachusetts, June 1997.

[2] Pree, W., Design Patterns for Object-Oriented Software Development, Addison-Wesley Publishing Company, 1995

[3] Fayad, M.E., Implementing Application Frameworks: Object-Oriented Frameworks at Work, Wiley Computer Publishing, 1999

[4] Biggerstaff, T. , Software Reuse , ACM Press 1989

[5] Mattsson, M. *Evolution and Composition of Object-Oriented Frameworks*, PhD Thesis, University of Karlskrona/Ronneby, 2000

[6]http://www.omg.org/technology/xml

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1995.

[8] Oliveira, T ,A Systematic Approach to Object Oeriented Framework Instantiation, PhD Thesis, Computer Science Department, (PUC-Rio) Brazil 2001

[9] Alencar P.S.C. , Cowan, D.D. Oliveira, T.C. , Lucena C.J. P. Process-Based Representation and Analysis of Framework Instantiation – Techincal Report University of Waterloo CS-2001-13

[10] Herbsleb J.D. , Mokckus, A. , Finholt T.A., Grinter R.E. , An Empirical Study of Global Software Development: Distance and Speed, p81-90, International Conference on Software Engineering, Toronto, Canada May 2001

[11]Krasner, G.E., Pope, S.T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming 1(3), 1988

[12] Froehlich, G., Hoover, H.J., Liu L. and Sorenson, P.G. Hooking into Object-Oriented Application Frameworks, Proc. 19th Int'l Conf. on Software Engineering, Boston, May 1997, 491-501

[13] Fontoura, M. A Systematic Approach to Framework Development. Ph.D. Thesis, Computer Science Department, (PUC-Rio), 1999.

[14] Ortigosa A., Campo M., Salomon R., *Towards Agent-Oriented Assistance for Framework Instantiation*. In Proc. OOPSLA '00, Minneapolis, Minnesota USA, ACM SIGPLAN Notices, 35, 10, 2000, 253-263

[15] Mens,K, Michiels,I , Wuyts,R, Supporting Software Development through Declaratively Codified Programming Patterns, In SEKE 2001 Proceedings, Knowledge Systems Institute, pp. 236-243, International conference on Software Engineering and Knowledge Engineering Buenos Aires Argentina June 13 -15 2001., 2001.

Three Musketeers to the Rescue
Meta-modelling, Logic Programming, and Explanation-based Constraint Programming for Pattern Description and Detection

Yann-Gaël Guéhéneuc[⋆]

École des Mines de Nantes
4, rue Alfred Kastler – BP 20823
44307 Nantes Cedex 3
France
guehene@emn.fr

1 Introduction

Software maintenance is a costly and tedious phase in the software development process [37]. During this phase, a maintainer needs both to understand and to modify a program source code. Therefore, she needs a representation of the program that accurately reflects its structure and its behavior. Then, she must find those places in the program that require modifications. Finally, she must perform changes that improve the program behavior and that do not introduce further defects.

In our research work, we focus on the maintainer's first and second tasks: The obtention of an accurate representation of the program structure and behavior, and the detection of places to improve. We propose a set of software engineering tools for the representation of Java programs (structural and dynamic information), and for the (semi-) automated detection of design patterns and design defects. Design patterns and design defects are related: We assume that a group of classes whose micro-architecture is similar (but not identical) to a design pattern corresponds to a possible design defect [16]. Either the pattern is distorted because it has been clumsily implemented, or the pattern is distorted because it does not fit in the architecture: In each case, the maintainer may want to analyze further the highlighted micro-architecture.

We concentrate on programs developed using object-oriented programming languages, because we are interested in detecting (object-oriented) design patterns and design defects. However, we believe we could apply our approach to any programming language (functional, procedural...) given suitable elements to describe patterns and programs written with this programming language.

⋆ This work is partly funded by Object Technology International, Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

We develop:

- PatternsBox, a tool to describe design patterns and design defects, to apply design patterns, and to detect complete forms of design patterns [1].
- Caffeine, a tool for the dynamic analysis of Java programs. We develop a library to analyze binary class relationships [18].
- Ptidej Solver, an explanation-based constraint solver [24] to detect design patterns and design defects [19].
- Ptidej, a tool to visualize program architecture and behavioral information, to generate the problems related to the detections of design patterns and of design defects, and to display results of the detections.

These tools use different declarative meta-programming paradigms. According to the Merriam-Webster's (2002), a paradigm is "*a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated*". We interpret this definition and we use the term declarative meta-programming to denote programs (and languages) which subjects of computation are programs or program artifacts:

- In PatternsBox, we describe patterns using a meta-model. The descriptions represent the structure and the behavior of the *Solution* elements of the patterns. Descriptions are first-class entities, which we can manipulate (to generate source code...) and reason about (to compare with other descriptions...).
- In Caffeine, we analyze a program execution using queries. We express the queries in term of a trace model and of an execution model of the program execution, using logic programming. The Caffeine system runs as a co-routine of the program to analyze, which emits events that abstract its runtime behavior. The Caffeine system receives, analyzes, and drives the program execution according to the queries.
- In Ptidej Solver, we use explanation-based constraint programming [24] to solve problems representing the detection of design patterns and design defects in a program architecture. A problem decomposes in a constraint system, generated from a pattern, and in a domain, representing the program architecture. The constraint system and the problem domain are generated automatically from the pattern and the program architecture, both expressed using the PatternsBox meta-model.
- In Ptidej, we display program architectures in terms of the PatternsBox meta-model and of behavioral information, obtained using Caffeine. We display classes, inheritance and implementation relationships, and binary class relationships, such as the association, the aggregation, and the composition relationships [17]. We visualize the results of the detection computed by the Ptidej Solver.

In this position paper, we present and motivate the use of the three distinct declarative meta-programming paradigms: Meta-modelling, logic programming, and explanation-based constraint programming. We also describe a succinct scenario highlighting the use of our tools.

2 Pattern Description

Patterns have been widely accepted by software practitioners. They cover all phases of the software development process: Requirements [20]; Analysis [12]; Architecture [6]; Design [15]; Implementation (idioms) [7]; Defects [5]; Refactoring [13]; Testing [14].

Requirements, analysis, and architectural patterns are high-level, informal, and general-purpose patterns; Whereas design, defect, and implementation patterns are tightly coupled with the software development process and implementation. Thus, it is desirable to formalize these patterns [2, 30, 32]. Formalization may support the reification, the instantiation, the application, the detection, and the comparison of patterns. Several techniques exist to formalize design, defect, and implementation patterns:

- Logic programming [27, 38].
- Logic-based notations [11].
- Meta-modelling [2, 25, 28, 34]
- Program generation [36].
- Program transformations [35].
- UML-based and associated notations (*eg.*, OCL) [15, 31, 34].
- State charts [15].
- Protocols and finite-state machines.

The choice of a particular technique depends on the intended use of the formalization. In PatternsBox, we want to reify design patterns and design defects as first-class entities from declarative descriptions of the patterns structure and behavior. Then, we want to reason and to interact with the patterns: To instantiate them; To display them (either as source code, as constraint systems, or as constraint domains...); And, to recognize them in source code. Therefore, we choose the meta-modelling technique to represent patterns.

Other authors use different techniques to formalize patterns. However, these techniques have shortcomings *w.r.t.* our intended use of the formalization. In a system based on logic programming, a set of predicates describes a pattern, but the pattern is not reified within the system: We cannot reason about it or interact with it from *within* the system. For example, code generation takes place in a separate module, which input is the set of predicates.

3 Dynamic Information

Program analysis is an important issue in object-oriented software engineering. Program analysis serves different purposes, such as extracting object-model from source code [22], understanding dependencies among classes and their instances [17], or verifying, refuting, simulating, and checking properties [21]

Program analysis may be performed either statically or dynamically. On the one hand, some information may be costly, complex, or even impossible to extract from static program-analysis. On the other hand, the results of dynamic analyses are valid only for the set of considered executions.

In CAFFEINE, we want to perform dynamic analyses of Java programs to extract information on the dependencies among classes and their instances. We model the execution of a program as a trace, which is a history of execution events. Execution events abstract the execution of the program as Prolog predicates, for example `fieldModification(...)`, `finalizerEntry(...)`, or `programEnd(...)`. We request the next available events through a Prolog engine, which runs as a co-routine of the program being analyzed. The Prolog engine drives the execution of the program under analysis using the Java platform debug architecture API [33]. Prolog already showed its adequacy to query traces in different works [9, 10].

For example, the following Prolog predicates count the number of times method `startTest` executes:

```
query(N, M) :-
    nextEvent(
        [generateMethodEntryEvent],
        E),
    E = methodEntry(_, startTest, _, _, _),
    N1 is N + 1,
    query(N1, M).
query(N, N).

main(N, M) :- query(N, M).
main(N, N).
```

First, we order CAFFEINE to obtain the next `methodEntry` event from the program execution, using the `nextEvent` predicate. Second, we filter out method entries corresponding to the `startTest` method, using the `=` predicate. Third, we increment a counter and recursively call the query. The use of logic programming allows a powerful and quite natural expression of queries on the trace of the program execution. In particular, we develop a set of predicates to verify properties on binary class relationships [17].

Other techniques to reason about program executions include universally quantified predicates [26], regular expressions, or temporal logic [8]. However, for our purpose, each one of these techniques has drawbacks. Universally quantified predicates are more declarative than Prolog queries, but they only deal with state information. Regular expressions are simpler and more efficient but with less power of expression. Temporal logic eases expressing temporal relationships (such as precedence) but is less expressive than logic programming, which offers a unification mechanism and high-level pattern matching capabilities.

4 Pattern Detection

The automated detection of patterns in source code is a difficult task and is subject of many works, such as [3, 4, 23, 29]. When patterns are used in a program architecture, there is no real links between the patterns (their actors and the relationships among them) and the source code (the classes and the relationships among them).

In PTIDEJ SOLVER, we use the information collected from PATTERNSBOX and CAFFEINE to represent a program architecture. From this information and given a pattern described using the PATTERNSBOX meta-model, we automatically generate a constraint problem where the constraints and the variables correspond to the pattern to detect, and where the domain correspond to the classes and the relationships among the classes of the program architecture.

We extend and use a constraint solver with explanations [19, 24], PALM, to obtain automatically all the complete and distorted solutions to the constraint problem, even if the constraint problem is over-constrained or if there exists no solution with all the constraints. Distorted solutions are solutions to a subset of the given constraints and thus represent possible design defects *w.r.t.* the design pattern being detected.

For example, the PALM code excerpt below instructs the constraint solver to compute solutions to the problem of *Good Inheritance*. The *Good Inheritance* pattern states that an entity `Super-entity` (class or interface) must not know[1] about any other entity `Sub-entity` that extends (or implements) it.

```
let pb := makePtidejProblem("Good Inheritance", length(listOfEntities), 90),
    superEntity := makePtidejIntVar(pb, "Super-entity", 1, length(listOfEntities)),
    subEntity   := makePtidejIntVar(pb, "Sub-entity", 1, length(listOfEntities)) in (
    post(pb, makeStrictInheritancePathConstraint(
             subEntity,
             superEntity),
             100),
    post(pb, makeIgnoranceConstraint(
             superEntity,
             subEntity),
             50))
```

First, we declare a new problem, which domain is the number of entities in the program architecture, `length(listOfEntities)`, and which maximum level of constraint relaxation is 90. Second, we declare the variables of the problem: Two variables `superEntity` and `subEntity`, which values range from 1 to `length(listOfEntities)`. Finally, we post two constraints:

- The first constraint, `StrictInheritancePath`, states that the two variables must instantiate such that the entity in variable `subEntity` extends (or implements) the entity in variable `superEntity`.
- The second constraint, `Ignorance`, states that the two variables must instantiate such that the entity in variable `superEntity` does not know about the entity in variable `superEntity`.

We assign a weight of 50 to the `Ignorance` constraint to allow the PTIDEJ SOLVER to remove this constraint when it fails to find more solutions. The solutions found without the `Ignorance` constraint corresponds to distorted solutions to the *Good Inheritance* pattern: These solutions are possible design defects.

[1] The precise definition of the knowledge relationship is out of the scope of this article, the interested reader may refer to work [17].

The use of explanation-based constraint programming makes it easy to express complex problems in terms of the solutions we want rather than how to compute the solutions. Also, it eases the explanation of distorted solutions and thus the detection of possible design defects.

Explanation-based constraint programming is more powerful than other approaches such as fuzzy logic [23] or logic programming [38]. Fuzzy logic proved its usefulness for detecting defects in class declarations, but generic fuzzy reasoning nets seem difficult to construct and they require fine tuning. Logic programming only helps in detecting classes whose relationships are described by the logical rules: It does not *directly* help in detecting distorted solutions.

5 Example

We now present a scenario highlighting the use and integration of our different tools: Caffeine, PatternsBox, Ptidej, and Ptidej Solver[2].

A maintainer desires to understand better the architecture of JUnit v3.7 and to find possible design defects. We assume that the maintainer starts from scratch, with no pattern described yet. We also assume that she has a good knowledge of the design patterns in [15].

She turns to Ptidej and she loads JUnit v3.7 to display its architecture. She quickly browses the program architecture and notices the `TestResult` class that possesses a container-aggregation relationship with the `TestListener` interface[3] (both classes from package `jtu.framework`). She wonders if this container-aggregation could, in facts, be a composition-container relationship.

She turns to Caffeine and writes a simple program that uses Caffeine to analyze the relationship between classes `TestResult` and `TestListener` with a specific Prolog query [17]. She runs her program with the `MoneyTest` test class, provided with JUnit v3.7, as input and she obtains the confirmation that the relationship between classes `TestResult` and `TestListener` is indeed a composition-container relationship.

She goes back to Ptidej and she loads the result of the dynamic analysis. The model of the architecture changes to reflect the behavioral information. She recognizes that such a container-composition relationship between two classes is the sign of a (possible) implementation of the Composite design pattern: She decides to verify this possibility. First, she builds a meta-entity describing the *Solution* element of the Composite design pattern, using constituents of the PatternsBox meta-model. Second, she uses PatternsBox to interact with the Composite meta-entity. She instantiates the meta-entity into an abstract model. She could parameterize the abstract model to fine-tune the model but she decides to go on with the abstract model as it is. She chooses the PaLM custom-constraint builder from the set of available builders and save the constraint system associated with the abstract model of the Composite design pattern to disk. Third,

[2] For the sake of place, we only summarize their use.

[3] For a discussion on binary class relationships, the interested reader may turn to [17].

she generates the domain corresponding to the architecture of JUnit v3.7 and calls the Ptidej Solver. The Ptidej Solver computes the set of complete and distorted solutions.

Finally, the maintainer loads the solutions to the constraint problem and browses the two different distorted solutions found by the constraint solvers. The solutions are, respectively, close at 60% and 1% to the micro-architecture advocated by the Composite abstract model. She must now further investigate the micro-architectures highlighted by the constraint results and decide whether or not these micro-architectures represents a Composite design pattern and whether or not modifications are required.

6 Conclusion

Declarative meta-programming is at the core of our software engineering tools. We conjointly use meta-modelling, logic-base programming, and explanation-based constraint programming to solve very practical software engineering problems: The declaration of patterns, the representation of programs, and the detection of patterns in the source code of programs.

Acknowledgements

The author deeply thank Hervé Albin-Amiot for the PatternsBox tool and its meta-model, and his kind support and invaluable advices.

References

[1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *Proceedings of ASE*, pages 166–173. IEEE Computer Society Press, November 2001.

[2] H. Albin-Amiot and Y.-G. Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of the ECOOP Workshop on Automating Object-Oriented Software Development Methods*. University of Twente, The Netherlands, October 2001. TR-CTIT-01-35.

[3] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. *Proceedings of the 6 th Workshop on Program Comprehension*, pages 153–160, 1998.

[4] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, University of Illinois at Urbana-Champaign, 1996.

[5] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. M. III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, Inc., 1998.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, Inc., 1996.

[7] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.

[8] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera specification language. Technical Report KSU CIS Technical Report 2001-04, Kansas State University, 2001. Submitted for journal publication.

[9] M. Ducassé. Coca: A debugger for C based on fine grained control flow and data events. In *Proceedings of ICSE*, pages 504–513. ACM Press, May 1999.

[10] M. Ducassé. OPIUM: An extendable trace analyser for Prolog. In *The Journal of Logic Programming, Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, volume 41, pages 177–223. Elsevier – North Holland, November 1999.

[11] A. H. Eden, A. Yehudai, and J. Y. Gil. Precise specification and automatic application of design patterns. In *Proceedings of ASE*, pages 143–152. IEEE Computer Society Press, November 1997.

[12] M. Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley Object Technology Series, 1996.
[13] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.
[14] E. Gamma and K. Beck. JUnit. Available at: `http://www.junit.org/`, 2002. Available at: `http://www.junit.org/`.
[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[16] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of TOOLS USA*, pages 296–305. IEEE Computer Society Press, July 2001.
[17] Y.-G. Guéhéneuc, H. Albin-Amiot, R. Douence, and P. Cointe. Bridging the gap between modeling and programming languages. Technical Report 02/09/INFO, École des Mines de Nantes, July 2002.
[18] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In *Proceedings of ASE*. IEEE Computer Society Press, September 2002.
[19] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design-patterns identification. In *Proceedings of the IJCAI Workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.
[20] A. Isazadeh, G. H. MacEwen, and A. J. Malton. Behavioral patterns for software requirement engineering. In *CD-Rom on CASCON*. Centre for Advanced Studies of IBM Toronto Laboratory and the Institute for Information Technology of the National Research Council of Canada, November 1995.
[21] D. Jackson and M. C. Rinard. Software analysis: A roadmap. In *Proceedings of ICSE, Future of Software Engineering Track*, pages 133–145. ACM Press, June 2000.
[22] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *Proceedings of ICSE*, pages 194–202. ACM Press, May 1999.
[23] J. H. Jahnke, W. Schäfer, and A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. *Proceedings of the European Software Engineering Conference*, pages 193–210, 1997.
[24] N. Jussien. e-Constraints: Explanation-based constraint programming. In *CP'01 Workshop on User-Interaction in Constraint Satisfaction*, December 2001.
[25] T. Kobayashi. Object-oriented modeling of software patterns and support tool. In *Proceedings of the ECOOP Workshop on Automating Object-Oriented Software Development Methods*. University of Twente, The Netherlands, October 2001. TR-CTIT-01-35.
[26] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In *Proceedings of ECOOP*, pages 135–160. Springer-Verlag, June 1999.
[27] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, 2002.
[28] B.-U. Pagel and M. Winter. Towards pattern-based tools. *Proceedings of EuropLop*, 1996.
[29] L. Prechelt and C. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–883, December 1998.
[30] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 2000.
[31] P. Rapicault and M. Fornarino. Instanciation et vérification de patterns de conception : Un méta-protocole. *Proceedings of LMO, in French*, pages 43–58, 2000.
[32] J. Soukup. *Implementing Patterns*, chapter 20. Addison-Wesley, 1995.
[33] Sun Microsystems, Inc. Java platform debug architecture, 2002. Available at: `http://java.sun.com/ products/jpda/`.
[34] G. Sunyé. *Mise En Oeuvre de Patterns de Conception : Un Outil*. PhD thesis, Université de Paris 6 – LIP6, July 1999.
[35] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection. *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98, Vancouver, Canada*, pages 56–60, October 1998.
[36] K. D. Volder. Implementing design patterns as declarative code generators. Submitted at ECOOP 2001, 2001.
[37] S. G. Woods, A. E. Quilici, and Q. Yang. *Constraint-Based Design Recovery for Software Reengineering – Theory and Experiments*. Kluwer Academic Publishers, Kluwer Academic Publishers Group, Distribution Center, Post Office Box 322, 3300 AH Dordrecht, The Netherlands, 1998.
[38] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. *Proceedings of TOOLS USA*, pages 112–124, 1998.

Using Declarative Metaprogramming To Detect Possible Refactorings

Tom Tourwé Johan Brichau* Tom Mens†
{tom.tourwe,johan.brichau,tom.mens}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050-Brussel-Belgium

July 19, 2002

Abstract

In this paper, we advocate the use of declarative metaprogramming to detect violations of important (object-oriented) design guidelines and best practices. This is particularly useful for detecting when a design should be refactored, and which refactorings in particular should be applied. As we will show, a declarative environment incorporating metaprogramming capabilities is very well suited for detecting such violations and providing information for possible refactorings.

1 Introduction

Many design guidelines and best practices have been proposed over the years, with the specific intent of promoting good object-oriented design principles [1, 10]. At the same time, much research has been devoted to identifying refactorings, e.g. high-level transformations, that can help in transforming an inflexible, lowquality design into a more flexible one [4, 9]. Although some primitive tools exist [5], recognizing when a design guideline is violated, and when refactorings may thus be necessary, remains a manual process, as is identifying which refactorings in particular could be used to remedy the situation.

In this paper, we advocate the use of declarative metaprogramming (DMP) [3, 11] to fill in this gap. The declarative nature of DMP allows us to accurately express design guidelines in a straightforward and intuitive way. Moreover, explicit metaprogramming enables us to reason about the source code of an application so as to actually verify whether it does not violate any of those guidelines.

In what follows, we will present an example of how declarative metaprogramming can be used to detect defective designs. We will achieve this by using the SOUL declarative metaprogramming environment [11], both as a medium to describe the conditions for defective designs, and as a test case for these conditions.

*Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)
†Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

2 Why Declarative metaprogramming

Design guidelines are actually rules that the implementation of an application should adhere to. Most of these rules are quite simple and can often be expressed in a straightforward way in natural language. The *Law Of Demeter* for instance, is stated as follows: *"an operation O of class C should call only operations of the following classes, called preferred supplier classes: the classes of the immediate subparts (computed or stored) of the current object, the classes of the argument object of O (including the class C itself) and the classes of object created by O"* [6]. Other rules are expressed in quite a similar way [7, 8]. In order to actively verify such rules, they have to be specified programmatically. This is quite cumbersome to achieve in current-day standard programming languages, such as C++ and Java. First of all, these languages are not particularly well suited to express *rules* per se. Second, most of the current-day standard programming languages do not have adequate meta-programming capabilities (with Smalltalk being the exception that proves the rule).

SOUL on the other hand, is a logic programming language that is tightly integrated with the standard (Smalltalk) development environment. Logic languages naturally allow to express rules, by mere definition. Furthermore, the tight integration allows SOUL to reason about and manipulate programs written in Smalltalk, and does make SOUL perfectly well suited for metaprogramming purposes. Moreover, it also means that the SOUL environment is always synchronized with the development environment. Another advantage of using SOUL is the declarative nature of the logic paradigm. It has already been shown that logic programming languages are particularly well suited for metaprogramming, because they allow meta programs to be specified in an intuive way [2].

The SOUL programming language is actually a Prolog-dialect with some extensions to allow Smalltalk expressions to be evaluated as part of a logic program. These extensions allow to reify and represent all information from the Smalltalk image as logic facts. SOUL comes with an extensive library of logic programs that reason about this information to conclude more high-level information, such as the existence of design patterns.

3 Detecting Design Guideline Violations

3.1 Inappropriate Interfaces

Good interfaces are extremely important when designing flexible and reusable object-oriented systems. Any situation in which the interface of a class is inappropriate, incomplete or unclear should thus be avoided at all costs.

As a concrete example, consider the `AbstractTerm` hierarchy depicted in Figure 1. This hierarchy shows part of the implementation of the SOUL environment. As can be observed, the `CallTerm, CompoundTerm, SmalltalkTerm` and `QuotedCodeTerm` classes each provide an implementation for the `terms` method, whereas all other classes (including `AbstractTerm`) do not. This situation creates a problem when we want to extend the `AbstractTerm` hierarchy with a new class. It is not directly clear from the design which subclasses of `AbstractTerm` should provide an implementation for the `terms` method, and which subclasses should not. A developer confronted with this situation should

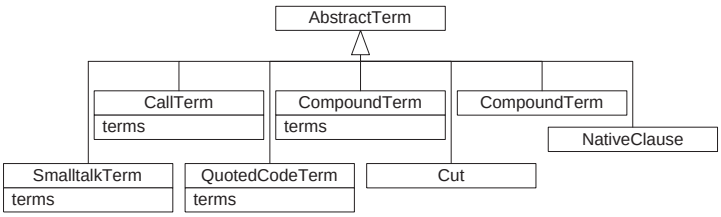

Figure 1: An example of an inappropriate interface

thus know exactly what he is doing.

To correct the design, two different solutions are possible. Either an intermediate superclass is inserted between the original superclass and all subclasses that share the interface. This newly introduced superclass should then provide the shared interface. Another option is to extend the interface of the original superclass with the interface shared by the subclasses. This also exposes the interface to subclasses that did not originally provide it, however, which may not be desired.

3.2 Problem Statement

The above mentioned problem occurs whenever some (but not all) of the subclasses of a class share an interface, that is not provided by that class itself. Detecting such situation manually is not as straightforward as it may seem, however. Standard browsers included in current-day programming environments only offer a local and narrow view of the source code. Developers thus often lack a more general overview, that would allow them to identify such problems. Appropriate tool support is thus clearly indispensable, and this is where declarative metaprogramming comes in.

3.3 Detecting The Problem

Using SOUL, we can easily detect this situation by implementing the following logic rules.

```
implementingSubclasses(?superclass,?selector,?subclasses) if
     subclassImplements(?superclass,?selector, ?),
     not(classImplements(?superclass,?selector)),
     findall(?subclass,
                subclassImplements(?superclass,?selector,?subclass),
                ?subclasses).
```

The *implementingSubclasses* predicate calculates all subclasses of a given superclass that implement a particular selector which is not implemented by the superclass itself. The rule is implemented in terms of two auxiliary predicates, *classImplements* and *subclassImplements*. The latter predicate is implemented as follows:

```
subclassImplements(?superclass,?selector,?subclass) if
     subclass(?subclass,?superclass),
     classImplements(?subclass,?selector)
```

It uses the *subclass* and *classImplements* predicates that are part of the library of logic rules in SOUL that consult the implementation to retrieve the requested information. The *subclass* predicate checks whether there exists a direct inheritance relation between two classes, while the *classImplements* predicate checks whether a class implements a particular selector.

What remains is verifying whether the set of subclasses that is calculated by the *implementingSubclasses* predicate does not contain all subclasses of the given class. This simply boils down to comparing sets for equality:

```
inappropriateInterface(?superclass,?selector,?subclasses) if
     implementingSubclasses(?superclass,?selector,?subclasses),
     not(allSubclasses(?superclass,?subclasses))
```

We can now use SOUL to detect inappropriate interfaces in our implementation. Therefore we invoke the following query, which will return the design violation we mentioned:

```
if inappropriateInterface([AbstractTerm],?selector,?subclasses)
```

3.4 Discussion

The above discourse clearly shows that declarative metaprogramming is very well suited to express the problem of inappropriate interfaces. Moreover, the rules presented above not only detect the situation of inappropriate interfaces, but also convey information about the interface that is shared by the subclasses, and those subclasses themselves. This can prove valuable when a particular refactoring has to be applied.

Using the above rules, we were able to identify several interface conflicts in the implementation of the SOUL environment. All reported conflicts were effectively real conflicts that needed to be solved in order to end up with a better and more suitable design. The information gathered by the logic rules was instrumental in applying the necessary refactorings.

We envision a programming environment where several of these 'design guidelines violations' can be detected using declarative metaprogramming. Depending on the detected violations, a range of refactorings can be proposed to the developer, who can choose the appropriate one to be applied. This linking of violations to the correct refactorings remains to be investigated, but once again, the declarative metaprogramming environment could prove to be ideal to express such information.

4 Conclusion

In this paper, we have shown the usefulness of a declarative meta-programming approach for detecting violations of important design guidelines and best practices. We demonstrated that the declarative nature of such an approach allows us to define the conditions under which such violations occur in a straightforward and intuitive way. Moreover, we illustrated that explicit metaprogramming capabilities are absolutely essential for such an approach.

While we have only shown one, rather simple, example of a design guideline violation, we believe the approach is general enough to detect all sorts of other violations as well, even on a much complexer scale. Further experiments in this

direction are mandatory, however. It is our firm believe that such an approach could be a first step towards tool support for detecting not only when a design should be refactored, but also which particular refactorings it should undergo.

References

[1] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[2] James R. Cordy and Medha Shukla. Practical metaprogramming. Technical report, Software Technology Laboratory, Queen's University, 1992.

[3] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1998.

[4] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison Wesley Longman, 1999.

[5] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. Int'l Conf. Software Maintenance*, pages 736–743. IEEE Computer Society Press, 2001.

[6] Karl Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Computer Society*, pages 38–48, 1989.

[7] Barbara H. Liskov and Stephen N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[8] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[9] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[10] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company, April 1996.

[11] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.

A Language-centric Approach to Software Engineering: Domain Specific Languages meet Software Components

Gopal Gupta

Department of Computer Science
University of Texas at Dallas
gupta@cs.utdallas.edu

Abstract

Domain Specific Languages (DSLs) are high level languages designed for solving problems in a particular domain, and have been suggested as means for developing reliable software systems. Component based software engineering has been proposed as a way of reducing the complexity of software development by providing ready-made *software components* for parts of the system being developed. We present an approach that amalgamates these two technologies. Our approach relies on a (constraint) logic programming-based framework for specification, efficient implementation, and automatic verification of domain specific languages (DSLs) around software components. Our framework allows the implementation infrastructure for a DSL (interpreter, compiler, debugger, and profiler) to be automatically obtained (in a provably correct manner) from the semantic specification of the DSL. Additionally, the semantic specification can be used for (semi-)automatically verifying programs written in the DSL as well as for automatically checking that *component contracts* are consistent with the manner in which they are used. This new framework is currently being applied for developing several DSLs, designed around software components. The most significant of these is a DSL, called ΦLog, being developed to enable biologists to program phylogenetic problems in biology.

1 Introduction

Writing software that is robust and reliable is a major problem that software developers and designers face today. Development of techniques for building reliable software has been an area of study for quite some time. Recently, two distinct approaches have been proposed:

- **Approaches based on domain specific languages (DSL):** In the DSL approach [5, 25, 14, 20, 22, 13], a domain specific language is developed to allow users to solve problems in a particular application area. A DSL allows users to develop complete application programs in a particular domain. Domain specific languages are very high level languages in which domain experts can write programs *at a level of abstraction at which they think and reason*. DSLs are not "general purpose" languages, rather they are supposed to be just expressive enough to "capture the semantics of an application domain" [20]. The fact that users are able to code problems at the level of abstraction at which they think and the level at which they understand the specific application domain results in programs that are more likely to be correct, that are easier to write, understand and reason about, and easier to maintain. As a net result, programmer productivity is considerably improved. The DSL-based approach can be regarded as a top-down approach, in which a simple, high-level language interface is presented to the programmer to ease the task of programming.

- **Approaches based on Component Based Software Engineering (CBSE)** In this approach [2, 3, 4, 29] a repository of ready-made software components is assumed. Programmers write "glue code" to put together existing software components to solve a particular problem. The "glue code" is written in a traditional programming language and makes use of components that can be regarded as ready-made library procedures. Use of components results in software-reuse: tasks that are similar in nature need not be programmed again and again. The components based approach also results in improved programmer productivity due to software reuse. If the library of components is standardized, the components based approach can also result in code that is easier to read and maintain (e.g., components can be regarded as plug-in modules whose implementation can change, and as long as the

interface of the component remains fixed, the software that uses components need not be changed). The components-based approach can be thought of as a bottom-up approach in which low level implementation details of those parts of the program—for which components are available—can be hidden from the programmer.

Both DSL-based and CBSE-based approaches have their advantages as well disadvantages. The two technologies of DSL and components can be synergistically used to create software faster and in a provably correct manner. We argue that combining the two overcomes some of the disadvantages of both approaches. We achieve this combination via a semantics-based approach that yields an efficient implementation infrastructure (interpreters, compilers, debuggers, and profilers) for DSLs. In this semantics-based approach, a *denotational semantics* [27] of the DSL is written in terms of software components available for that application domain (components are, in fact, treated as part of the *semantic algebras* [27] of this semantics). This semantics is coded using *Horn Logic* (or pure Prolog) and *Constraints* [28], and is executable [17]. The executable semantics yields an interpreter that may make calls to the various components. The executable semantics can be extended in a simple way to obtain debuggers and profilers. Given a program P written in the DSL, the semantic interpreter of the DSL can be partially evaluated [21] w.r.t. P to obtain "compiled code" in terms of calls to software components. Because the interpreter, compiler, debugger and profiler are all derived from the semantic specification, they are provably correct and obtained automatically from the semantic specification of the DSL. Also, the time taken for each iteration of the DSL design is much less, as changing the DSL only requires making change to its semantics, the modified implementation infrastructure (i.e., the interpreter, compiler, debugger, profiler) can be automatically derived from this semantics. Defining the semantics of DSL in terms of components, makes the task of specifying this semantics easier. Finally, component contracts can be specified as constraints [15], their consistency w.r.t. components' use can be checked in the semantic interpreter.

2 Domain Specific Languages (DSL)

The task of developing a program to solve a specific problem involves two steps. The first step is to devise a solution procedure to solve the problem. This steps requires a domain expert to use his/her domain knowledge, expertise, creativity and mental acumen, to devise a solution to the problem. The second step is to code the solution in some executable notation (such as a computer programming language) to obtain a program that can then be run on a computer to solve the problem. In the second step the user is required to map the *steps* of the solution procedure to *constructs* of the programming language being used for coding. Both steps are cognitively challenging and require considerable amount of thinking and mental activity. The more we can reduce the amount of mental activity involved in both steps (e.g., via automation), the more reliable the process of program construction will be. Not much can be done about the first step as far as reducing the amount of mental activity is involved, however, a lot can be done for the second step. The amount of mental effort the programmer has to put in the second step depends on the "semantic" gap between the level of abstraction at which the solution procedure has been conceived and the various constructs of the programming language being used. Domain experts usually think at a very high level of abstraction while designing the solution procedure. As a result, the more low-level is the programming language, the wider the semantic gap, and the harder the user's task. In contrast, if we had a language that was right at the level of abstraction at which the user thinks, the task of constructing the program would be much easier. A domain specific language indeed makes this possible.

However, a considerable amount of infrastructure is needed to support a DSL, a major disadvantage of this approach. First of all, the DSL should be manually designed. The design of the language will require the inputs of both computer scientists and domain experts. Once the DSL has been designed, we need a program development environment (an interpreter or a compiler, debuggers, editors, etc.) to facilitate the development of programs written in this DSL. The implementation infrastructure of the DSL (i.e., its compilers, debuggers, profilers, etc.) will constantly change as the language evolves. Making changes to the implementation infrastructure is a daunting task, and we believe is a major hurdle to the DSL-based approach being widely employed. Leveson et al [22] have used the DSL based approach for designing software for airplane control. However, they observe that the design of the DSL can take as much as 3 years [22]. We believe that this is primarily because of the reason that a language is fully understood only after its implementation infrastructure (interpreter, compiler, etc.) has been developed and used for writing

and executing a few programs by the domain experts. Developing the implementation infrastructure, or modifying and changing it takes a long time, resulting in each design iteration of the DSL taking a long time as well.

3 Software Components

A software component [29, 2, 3, 4] is a unit of independent deployment that has no persistent state and that may have been developed by a third party. Software components can be thought of as software modules that have been developed for commonly encountered tasks and that can be employed in any software system when needed. Components have contractually specified interfaces and explicit context dependencies only [29]. The advantage of software components is they can be bought from third party, and can be freely reused. Repositories of software components have been developed for use in software development projects to reduce the programming effort involved. The main advantage of software components is they facilitate software reuse, and thus can considerably reduce programmers' burden. In the software components based approach components are composed together using "glue code" written in some traditional language.

The CBSE approach frees a programmer from reprogramming many tasks. However, many problems still arise or remain. CBSE does not completely free the programmer from low-level programming since the components still have to be glued together in a low-level way. Essentially, all tasks in a software system for which components cannot be found still have to be programmed in a low level way. Finding components from a component repository that are suited to one's software needs is also a difficult task as component repositories can be quite large. Also, in the CBSE approach system integration has to be brought to the forefront of the software development process (typically it's at the end of the software development phase) and continually managed. The hardest problem in the CBSE approach, we believe, is knowing which components to use in a system, since pre-existing set of components may have been written for a pre-existing, possibly unknown, set of requirements (specified in component contracts [29]). These requirements may be very general, in which case the requirements of the system to be built will have to be made to conform to these general requirements, or the requirements with which the component is written may be quite restrictive and may fundamentally conflict with the requirements of the system in which this component is needed.

4 Domain Specific Languages and Software Components

In this position paper we espouse an approach that makes use of both DSLs and software components thereby producing a framework in which, we believe, software can be developed faster and more reliably. Our thesis is that components should be embedded in a domain specific language once and for all and this DSL should then be used by developers for writing applications. This is in contrast to having developers use components directly in their applications. The DSL can be thought of as a high-level wrapper language built around a set of components that are likely to be used in an application domain. The task of matching the requirements of the components and their suitability becomes the responsibility of the DSL designer(s) during language design, rather than of every application developer who uses components. Essentially, the language designer provides a proper abstraction for the components in the DSL itself in a cohesive way, freeing the developer from having to deal with the vagaries of a component's interface. The application developer only has to master the DSL, and thus avoids having to struggle with understanding the component's interface.

We have developed a semantics-based approach for combining DSL with components which works as follows. An application domain in which problems need to be solved is identified. A high-level domain specific language is designed so that domain experts can write applications at their level of abstraction. The semantics of this domain specific language is denotationally specified in terms of software components available for that domain. These software components have to be identified by the language designer(s). The semantics is coded using Horn Logic (pure Prolog) [28, 17] and is executable. The *executable semantics automatically yields an interpreter for the DSL* (this interpreter will call the various components during execution of a DSL program). The executable semantics can be extended in a simple way to obtain debuggers and profilers [19]. This is possible because the declarative semantic of the DSL explicitly passes the execution state as an argument of a predicate; hooks can be added after each call to semantic predicates that cause execution to pause and exhibit the execution state just as a debugger would. Likewise hooks can be added that record and compile execution statistics just as a profiler would. Given a program P written in the DSL, the semantic interpreter of the DSL can be partially evaluated [21] w.r.t. P (using partial evaluators

for Prolog such as Mixtus [26]) to obtain "compiled code" in terms of calls to these software components [17, 15].

A complete description of the framework as well as examples illustrating our approach [17, 19, 18] can be found elsewhere and are not included here due to lack of space. Our approach is currently being applied to develop a domain specific language called Φ-log [24] for allowing biologists to program phylogenetic applications. Φ-log is being implemented on top of software components developed by computational biologists for solving problems in phylogenetics. Our approach also being applied to develop a DSL for e-commerce applications.

5 Advantages of the Framework

Our framework combining DSLs and CBSE eliminates many of the disadvantages associated with software components and DSLs. The availability of components makes the design and specification of DSL much faster. The DSL in turn acts as a high-level "wrapper" around software components, freeing individual application developers from having to worry about potential mismatches between the interface of software components and the applications.

The principal advantage of combining DSL and software components via a semantics-based approach is that that the implementation infrastructure (interpreter, compiler, debugger, profiler) for a DSL can be rapidly prototyped. This can considerably speed up the iterative design-implement-modify-reimplement cycle involved in DSL design, thus removing what we believe to be one of the major hurdles that has precluded widespread use of DSL technology.

A semantic-based approach also facilitates development of DSLs based on software components available in a particular domain. Thus, a software engineering expert can look at a components repository, identify a set of software components in a particular domain and then design a DSL around these components to facilitate programming of tasks in that domain. The Horn logical semantics of the DSL will be written in terms of this set of components to obtain the implementation infrastructure for this DSL.

Thus, our approach can be applied in two ways: (i) in a top down manner, where an application domain is identified, a DSL designed, and its implementation infrastructure obtained around a set of software components using our semantics based framework; or, (ii) in a bottom up fashion, i.e., a set of closely related software components in a large repository is identified (in consultation with the users), and then a DSL designed around these software components, and its implementation infrastructure obtained. Note that in both cases, identification of the components and the design of the DSL will be collaboratively done by computer experts and the domain experts, however, the implementation infrastructure will be developed by the computer expert alone.

Once the design of a DSL has been fixed, and its implementation prototyped, highly efficient implementations can be obtained by implementing optimizing compilers for the DSL using the traditional compiler technology.

6 Language-centric Software Engineering

It can be argued that any complex software system that interacts with the outside world defines a domain specific language. This is because the input language that a user uses to interact with this software can be thought of as a domain specific language. For instance, consider a file-editor; the command language of the file-editor constitutes a domain specific language. This language-centric view can be quite advantageous to support the software development process. This is because the *semantic specification of the input language of a software system is also a specification of that software system*—we assume the semantic specification also includes the syntax specification of the input language. *If the semantic specification of the input language is executable, then we obtain an executable specification of the software system*. Note that this semantic specification can be given in terms of software components (i.e., s/w components are treated as primitive operations in the semantic specification). The preceding observations can be used to design a language semantics based framework for specifying, (efficiently) implementing using s/w components, and verifying (rather model checking or debugging in a structured way) software systems. The syntax and semantics of the input language is specified using Horn logic/Constraints, and is executable. Efficient, provably correct, compilation of the software systems in terms of software components can be obtained via partial evaluation. The resulting executable specification can also be used for verification, model checking

and structured debugging. Thus, in the light of the above discussion, software design is seen as the task of designing an input language. Implementation is seen as specifying the semantics of this input language in terms of software components.

An obvious candidate framework for specifying the semantics of a domain specific language is denotational semantics [27]. Denotational semantics has three components: (i) syntax, which is typically specified using a BNF, (ii) semantic algebras, or value spaces, in terms of which the meaning is given (in our framework software components will be treated as a semantic algebras), and, (iii) valuation functions, which map abstract syntax to semantic algebras. In traditional denotational definitions, syntax is specified using BNF, and the semantic algebra and valuation functions using λ-calculus. There are various practical problems with the traditional approach: (i) the syntax is not directly executable, i.e., it does not immediately yield a parser, (ii) the semantic specification cannot be easily used for automatic verification or model checking. Additionally, the use of separate notations for the different components of the semantics implies the need of adopting different tools, further complicating the process of converting the specification into an executable tool. Verification should be a major use of any semantics, however, this has not happened for denotational semantics; its use is mostly limited to studying language features, and (manually) proving properties of language constructs (e.g., by use of *fixpoint induction*). In our framework, we use Horn logic (or pure Prolog) for expressing denotational semantics as this facilitates the specification, implementation, and automatic verification/debugging of DSL programs, all in one framework. In the Horn logical denotational semantics framework, the BNF grammar can be specified as a *definite clause grammar* (syntax specification), which automatically yields a parser (executable syntax). The semantic algebras are defined in terms of software components and pure Prolog while the valuation functions (rather valuation predicates) are defined using pure Prolog. The valuation predicates can be executed on a Prolog interpreter yielding an executable semantics.

7 Applications of the Framework

Our logic programming based approach to software engineering is being applied to solve a number of problems. We are currently designing a domain specific language to enable biologists to program solutions to *phylogenetic inference problems* [24]. Phylogenetic inference [10] involves study of the biocomplexity of the environment based on genetic sequencing and genetic matching. Solving a typical problem requires use of a number of software systems—Genbank [6] (a repository of genetic data), BLAST [7] (a program for querying genetic databases), CLUSTAL W [8] (a program for aligning molecular sequences), PHYLIP [12] and PAUP [9] (both are programs for inferring evolutionary paths), etc., along with a number of manual steps (e.g., judging which sequence alignment for two genes is the "best"), as well as extra low-level coding to glue everything together. A biologist has to be considerably sophisticated in use and programming of computers to solve these problems, as outputs from various software systems have to be massaged and transformed, and then fed to other software systems. We are developing a DSL for phylogenetic inference that will allow biologists to program such interactions at a very high level, essentially allowing them to write/debug/profile programs at their level of abstraction. The semantic specification of this DSL is given in terms of available software components for phylogenetic inference (Genbank, CLUSTAL W, BLAST, PHYLIP, PAUP, etc). The task of solving phylogenetic problems will become much simpler for the biologist, giving them the opportunity to become more productive as well as be able to try out different "what-if?" scenarios.

Our approach is also being used to facilitate the navigation of complex web-structures (e.g. tables and frame-based pages) by blind users (blind-users typically access the WEB using audio-based interfaces). Given a complex structure, say a table, the web-page designer may wish to communicate only the essential parts of the table to a blind-user. In our approach, the web page-writer (or a third party) will attach to the web-page a domain specific language program that encodes the table navigation instructions [23].

Finally, our approach is also being used to generate provably correct code for SCR [1] specifications, and for developing a domain specific language for writing e-commerce applications.

8 Formal Verification

Automated or semi-automated verification is also possible in our semantics-based framework. Component contracts can also be enforced in the same framework. The semantic specification of a DSL $\mathcal{L}$ coded in Horn logic can be viewed as an *axiomatization* of the language constructs of $\mathcal{L}$. The denotation of a

program $\mathcal{P}_L$ written in $\mathcal{L}$ w.r.t. the Horn logical semantics of $\mathcal{L}$, $\mathcal{P}_d$, can be thought of as an axiomatization of the logic implicit in the program $\mathcal{P}_L$ or as an axiomatization of the problem that $\mathcal{P}_L$ is supposed to solve. This axiomatization can be used in conjunction with a logic programming engine to perform verification. Additionally, the relational nature of logic programming allows for the state space of a program written in $\mathcal{L}$ to be explored with ease. This fact can also be exploited to debug/verify properties of programs.

One standard way of gaining more confidence is to prove interesting properties about $\mathcal{P}_L$ but instead of using $\mathcal{P}_L$ we use its Horn logical denotation, $\mathcal{P}_d$, instead. Thus, given a property Φ that we want to prove about $\mathcal{P}_L$, we show that Φ is a logical consequence of axioms in $\mathcal{P}_d$, i.e., $\mathcal{P}_d \models \Phi$. However, note that the program may have been written under certain assumptions that were made regarding the input to the program.

Let us assume that we have a precondition I on inputs, $\bar{X}$, and a postcondition O on outputs, $\bar{Y}$, of the program $\mathcal{P}_L$. This means that for program $\mathcal{P}_L$, if $I(\bar{X})$ is true and $\mathcal{P}_L(\bar{X})$ terminates and produces outputs $\bar{Y}$, then $O(\bar{Y})$ must be true if $\mathcal{P}_L$ is correct. Let us assume that $\texttt{main\_p}(\bar{X}, \bar{Y})$ represents $\mathcal{P}_L$'s Horn logical-denotation. Then, the formula

$$(\mathcal{I}(\bar{X}) \wedge \texttt{main\_p}(\bar{X}, \bar{Y}) \rightarrow \mathcal{O}(\bar{Y})) \tag{1}$$

must hold true. Alternatively, the formula

$$(\mathcal{I}(\bar{X}) \wedge \texttt{main\_p}(\bar{X}, \bar{Y}) \wedge \neg\mathcal{O}(\bar{Y})) \tag{2}$$

must be false. We can use the formula above as a query to an LP system on which the logic program $\mathcal{P}_d$ has been loaded. If the program is correct (i.e., program terminates and the postconditions hold) then the above query would fail. Note that if components are being used as part of the semantic algebra in defining the semantics, then preconditions and postconditions that component satisfies may have to identified for each component and coded as logic/constraint programs. It is customary to specify *component contracts* via pre-conditions and postconditions. These preconditions and postconditions will be used during the verification of the program that employs these components in the above query.

If component contracts are specified as constraints, i.e., both preconditions and postconditions of a component $\mathcal{C}$ are specified as constraints, then these preconditions can be directly inserted into the denotation of the program that uses $\mathcal{C}$. If these constraints are consistent with the rest of the program denotation, then the constraints comprising the postconditions can be conjoined with the program denotation to complete the consistency proof. Note that use of constraints and logic programs for enforcing component contracts have also been advocated by others [15, 11], however, our approach embeds them in a semantics-based framework.

9 Conclusion

In this position paper we presented a framework that combines on domain specific languages with software components. In this framework, Domain Specific Languages are built around software components for rapid design, and their implementation infrastructure rapidly realized using a semantics based approach. The semantics based approach also permits automatic verification as well as generation of provably correct code. Note that the semantics and logic programming based approach is transparent to the end-user, i.e., the end-user only writes a program in the DSL, and is completely unaware of the underlying implementation technology used. We believe that using the semantics and logic programming based approach DSLs can be rapidly designed around collections of components in a particular domain, reducing the cost of prototyping and software development.

Acknowledgments The authors wish to thank Neil Jones, Enrico Pontelli, and Kishore Kamarupalle for helpful input. The author has been partially supported by grants from the NSF.

References

[1] C. L. Heitmeyer, R. Jeffords, B. Labaw. Automated Consistency Checking of Requirements Specification. ACM TOSEM 5(3):231-261. 1996.

[2] K. Bergner, A. Rausch, M. Sihling. Componentware—the Big Picture. 20th ICSE Workshop on Component-based Software Engineering. 1998.

[3] M. Zaremski, J. M. Wing. Specification Matching of Software Components. ACM TOSEM. 6(4):33-369. 1997.

[4] R. Schmidt and U. Assman. "Concepts for developing components-based systems." 20th ICSE Workshop on Component-based Software Engg., Japan, 1998.

[5] J. Bentley. Little Languages. *CACM*, 29(8):711-721, 1986.

[6] GenBank Overview. `www.ncbi.nlm.nih.gov/Genbank`.

[7] S.F. Altschul and B.W. Erickson. Significance of nucleotide sequence alignments: a method for random sequence permutation that preserves dinucleotide and codon usage. *Mol. Biol. Evol.*, 2:526–538, 1985.

[8] D. G. Higgins, J. D. Thompson, and T. J. Gibson. Using CLUSTAL for multiple sequence alignments. *Methods in Enzymology*, 266:383–402, 1996.

[9] D. L. Swofford. PAUP: phylogenetic analysis using parsimony, version 3.1.1. TR, Illinois Natural History Survey, 1993.

[10] D.L. Swofford, G.J. Olsen, P.J. Waddell, and D.M. Hillis. Phylogenetic inference. In David M. Hillis, Craig Moritz, and Barbara K. Mable, editors, *Molecular Systematics*, chapter 11, pages 407–514. Sinauer, Sunderland, MA, second edition, 1996.

[11] A. Cernuda del Rio, J. E. Labra Gayo, J. M. Cueva Lovelle. Applying the Itacio Verification Model to a Component-based Real-Time Sound Processing System. *Proc. of 2nd International Workshop on Logic Programming and Software Engg.*, 2001, Paphos, Cyprus.

[12] J. Felsenstein. PHYLIP: Phylogeny inference package, version 3.5c, 1993. see http://evolution.genetics.washington.edu/phylip/software.html, 2000.

[13] W. Codenie, K. De Hondt, P. Steyaert and A. Vercammen. From custom applications to domain-specific frameworks. In *Communications of the ACM*,Vol. 40, No. 10, pages 70-77, 1997.

[14] C. Consel. Architecturing Software Using a Methodology for Language Development. In *Proc. 10th Int'l Symp. on Prog. Lang. Impl., Logics and Programs (PLILP)*, Springer LNCS 1490, pp. 170-194, 1998.

[15] S. M. Daniel. An Optimal Control System based on Logic Programming for Automated Synthesis of Software Systems Using Commodity Objects. *Proc. Workshop on Logic Prog. and Software Engg*. UK, July 2000.

[16] L. King, G. Gupta, E. Pontelli. Verification of a Controller for BART: An Approach based on Horn Logic and Denotational Semantics. In *High Integrity Software Systems*. Kluwer Academic Publishers.

[17] G. Gupta. Horn Logic Denotations and Their Applications. In *The Logic Programming Paradigm: The next 25 years*, Proc. Workshop on Strategic Research Directions in Logic Prog., LNAI, Springer Verlag, May 1999.

[18] G. Gupta. Logic Programming based Frameworks for Software Engineering. *Proc. Workshop on Logic Programming and Software Enginering*. London, UK, July 2000.

[19] G. Gupta and E. Pontelli. Specification, Implementation, and Verification of Domain Specific Languages: A Logic Programming-based Approach. Essays in honor of Bob Kowalski. Spriger Verlag, 2001 (to appear).

[20] P. Hudak. Modular Domain Specific Languages and Tools. In *IEEE Software Reuse Conf*. 2000.

[21] N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503, 1996.

[22] N. G. Leveson, M. P. E. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Software Engineering - ESEC/FSE*, Springer Verlag, pages 127-145, 1999.

[23] E. Pontelli, W. Xiong, G. Gupta, A. Karshmer. *A Domain Specific Language Framework for Non-Visual Browsing of Complex HTML Structures* ACM Int. Conference on Assistive Technologies, 2000.

[24] E. Pontelli, D. Ranjan, G. Gupta, and B. Milligan. PhyLog: A Domain Specific Language for Describing Phylogenetic Inference Processes. In *Proc. First IEEE Computer Society Bioinformatics Conference*. Aug. 2002. (to appear). `http://www.cs.utdallas.edu/~gupta/phylog.ps`.

[25] C. Ramming. *Proc. Usenix Conf. on Domain-Specific Languages.* Usenix, 1997.

[26] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* Ph.D. Thesis, Royal Inst. of Techn. Sweden, 1994.

[27] D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W. C. Brown Publishers, 1986.

[28] L. Sterling and S. Shapiro. *The Art of Prolog.* MIT Press, 1996.

[29] C. Szyperski. Component Software: Beyond Object-oriented Programming ACM Press, New York. 1998.

A Declarative Persistency Definition Language

Muna Matar, Institute for Continuing Education
Ghent University, Belgium
Koenraad Vandenborre
Inno.com cva Belgium
Ghislain Hoffman, Herman Tromp
Department of Information Technology, Ghent University, Belgium

Abstract

This paper presents the Persistency Definition Language (PDL). PDL is a declarative language that is embedded in Javadoc style tags in order to introduce persistency metadata information in Java classes.

Introduction

Persistency is an object-oriented programming technique that deals with the ability of objects to exist beyond the lifetime of their creator or user. Persistency is usually implemented by preserving the state (attributes) of an object during its lifetime in a database, in most cases possibly in a relational database (RDBMS). The RDBMS technology is robust and widespread.

Designing software to connect an object-oriented software system to a relational database is a tedious task. Object structure and the table based relational technology are two widely differing paradigms. Hence, bridging the gap between the two worlds by mapping objects to database tables is often a task that takes a lot of effort, and is often too pervasive at the different layers of a multi-tiered architecture [2] [3].

Java offers introspection to obtain information about objects and classes at run-time. Unfortunately, introspection, although a powerful feature, is limited to gathering information about class structure and attribute values.

Meta information about classes that is related to making instances of those classes persistent can not be found from class code. Persistency related issues are impossible to declare in Java. This shortcoming of the language lead us to come up with another "declarative language" that helps to identify and declare persistency related information. This language is called the **P**ersistency

Definition **L**anguage or simply, **PDL**. PDL integrated with the introspection feature in Java provides a complete persistent description of persistent classes.

Related work

The need to provide auxiliary information for program elements appears to be growing. Information about fields, methods and classes as having particular attributes that indicate they should be processed in special ways by development tools, deployment tools, or run-time libraries is called annotations metadata [5]. The JSR 175, a metadata facility for the Java programming language [4] and the Sun Java Data Objects (JDO) [5] discuss this issue in details.

To describe how persistence related issues and information can be decoupled from class implementation, the Aspect Oriented Programming paradigm (AOP) can be used [7]. See also (http://aspectj.org).

The work of K. Vandenborre and others, described in [6], gives a description of a general methodology which illustrates how persistence can be made orthogonal from the class library by using the AOP.

What AOP presents in terms of solving some issues related to persistency is very valuable. But, even with using AOP, we still need a declarative mechanism by which we can introduce persistency related information into class code which AOP does not provide. That is why the need for PDL was still valid.

PDL

When working with persistency related issues like building a framework to store Java objects, we were confronted with the weak declaration mechanism in Java.

The main problem that we were faced with when using introspection and the reflection API was that declaring many of the persistency issues was not possible in Java. Java is not declarative enough to do so. Java allows only transient and non-transient to be declared and there is no way to incorporating detailed persistence information.

PDL was developed in the context of a storage framework called **PDLF**. The PDLF is an all Java object-relational declarative framework that enables Java objects to be persisted to relational databases. This framework provided persistency description of classes as well as the capability for objects to be stored and retrieved from a relational database. The full description of this storage framework is out of the scope of this paper and can be found in [1].

As mentioned above, PDLF makes Java objects capable of storing and retrieving themselves from a relational database. And since the gap between Java objects and the table based relational structure is very wide, a detailed persistency description of classes and attributes had to be provided somehow to the PDLF to help map objects into relational tables and therefore make objects capable of being made persistent.

An example of one of the persistency issues needed that could not be declared in Java, would be how to identify an object accessor. An Object accessor is an attribute of the object that can be used to access objects with in the database i.e query the database with. Normally an object accessor is an attribute of the class. When an attribute is declared in any Java class, using introspection, certain information about the nature of the attribute and the dynamic accesses to it (e.g. name, type, value, etc.) can be found. This kind of information indicates nothing about a possible use of the attribute in a certain application.

PDL was introduced to cover the shortcomings of Java in terms of persistency aspects. It helps classify persistency aspects as well as provide a persistency description of a class. The main advantage of PDL is to decouple the persistency aspects and description of a class from the class implementation.

PDL is a tagged based extensible language. This means that all declarations are done through using PDL tags. PDL tags are Javadoc style tags that are added to the source code of every persistent class.

PDL can be considered to be a Domain Specific Language (DSL) [8] in its general purpose and form. It is an embedded and a declarative language. Implementing DSLs as embedded languages is a well known technique.

PDL Tags

As described above, PDL is a descriptive tagged based language. It makes use of tags which are Javadoc style tags. Identifying what tags needed came from identifying what persistency aspect were needed and were not possible to cover using Java. After an extensive search and running some tests on some applications, twelve PDL tags were introduced. Those tags are: *@database*, *@table*, *@major*, *@minor*, *@persistent*, *@state*, *@accessor*, *@unique*, *@index*, *@contained*, *@size*, and *@compType*.

PDL tags can be classified as follows:

- versioning tags: which are tags that define the class version. They consist of the *@major* and *@minor* tags.
- mapping tags: which are tags that have to do with mapping classes and attributes to database tables. They consist of the *@persistent*, *@database*,

and *@table* tags.

- retrieval tags: which are tags used in queries. They include the *@accessor* tag.
- internal state tags: which are tags that describe the internal structure of objects. They consist of the *@state*, *@size*, *@contained*, and *@compType*.
- table design tags: which are tags that help in designing table columns. They include the *@unique* and *@index* tags.

Since defining new needed tags is the core of PDL, we can say that PDL is closely related to XML. In XML one can define his/her own set of tags. PDL lets one identify persistent aspects of classes using meaningful tags and it lets one add information (meta-data) about each aspect. PDL is also flexible in the sense that new needed tags can be defined and added. We have to keep in mind that PDL was developed to add persistency meta information to classes so any new added tags must be relevant to the purpose of finding PDL.

A detailed description of each of the PDL tags can be found in [1].

Using PDL Tags

PDL tags are added to the source file of any persistent class. They are embedded within a Javadoc comment. And since PDL tags are of Javadoc style, they are situated preceding attributes they provide persistency aspects to. Those tags act like trigger points in the source code. In addition to that, those tags provide a structure that can be processed by a special tool.

Example 1 below illustrates the way PDL tags are inserted into the source code of any persistent class.

Example 1:

```
package MyApplication;

    /**
     * @database "Company"
     * @table "Employee"
     * @major 01
     * @minor 00
     */
public class Employee {
      public static ClassVersion classVersion = new ClassVersion("01","00");

    /**
     * @persistent
```

```
     * @accessor
     * @index
     * @unique
     */
    private Name empName;

    /**
     * @persistent
     * @contained
     */
    private Address address;

    /**
     * @persistent
     * @accessor
     * @index
     * @size 10
     */
    private ByteField jobTitle;

    //constructor
    public Employee(){
    }

//other constructors and methods go here
}
```

Processing PDL Tags

To parse the PDL tags in the source files, a tool (PDL processor) has been developed. This tool uses a special Javadoc doclet. The tool takes a source file with PDL tags in it and produces an XML file. This XML file contains a persistency description of the persistent class.

It must be clear here that Javadoc and XML are used in the PDLF context merely as tools. XML as the universal format for structured documents and data on the Web is modular, easy to read and most important easy to parse. Through parsing an XML document an application can make the data available in different formats.

The PDL processor mentioned above makes use of an XML parser which helps generate SQL code that is used by the PDLF for different purposes such as creating database tables and storing and retrieving objects from those tables.

Running Javadoc on the source code presented above in example 1 with the special doclet (PDL processor) will produce the following XML file.

```
<?xml version='1.0'?>
```

```
<!DOCTYPE ClassLibrary >
 <classDescriptor Class="MyApplication.Employee">

   <classVersion major="01" minor="00">
   </classVersion>

   <db database="Company " table="Employee">
   </db>

 <pAttribute accessor="true" index="true" unique="true" contained="false">
       <attributeOfClass>
         MyApplication.Employee
       </attributeOfClass>
       <attributeName>
         empName
       </attributeName>
 </pAttribute>

 <pAttribute accessor="false" index="false" unique="false" contained="true">
       <attributeOfClass>
         MyApplication.Employee
       </attributeOfClass>
       <attributeName>
         address
       </attributeName>
 </pAttribute>

 <pAttribute accessor="true" index="true" unique="false" contained="false">
       <attributeOfClass>
         MyApplication.Employee
       </attributeOfClass>
       <attributeName>
         jobTitle
       </attributeName>
       <size size="10">
       </size>
 </pAttribute>

</classDescriptor>
```

Obtaining the XML file was one of two stages any persistent class needed to go through to be able to register with the PDLF. When a class registers with the PDLF, instances of this class can be made persistent in addition to the fact that a complete persistence description of that class is added to the central repository of the PDLF. This repository is a metadata container of persistent classes and their mappings to database tables. This repository is heavily used when reading and writing objects to database tables.

Conclusion

What we have presented in this paper is an extensible langauge, PDL, which can be used to declare persistency metadata information in classes. Due to the extensibility nature of this language, it can be also used to introduce other meta information about classes beside persistency class information.

It is important to note here that PDL emphases the idea of decoupling persistency issues from class implementation issues. It provides a single point of reference to persistency related information within a persistent class.

Providing this information using Javadoc style tags made it easy for us to present a persistent description of classes and process it using many of the available tools and technologies like XML.

References

[1] Muna Matar. A methodology for object persistence in Java based on a declarative strategy, Ph.d Thesis., Ghent Univesity, 2001.

[2] Peter Kroha. Objects and Databases. McGraw-Hill Publishing Company, 1993.

[3] S. Agarwal, A. Keller., and R. Jensen., Bridging object-oriented programming and relational databases. 1993

[4] Java Community Process, JSR 175, www.jcp.org/jsr/detail/175.prt

[5] Java Data Objects, JSR 12. Craig Russell. Sun Microsystems Inc.

[6] K. Vandenborre., M. Matar., and G. Hoffamn. Orthogonal persistence using Aspect Oriented Programming. The proceedings of the first AOSD workshop on Aspects, Components and Patterns for infrastructure software, April 2002.

[7] G. Kiczales, J. Lamping, A.Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programing. The proceddings of the European Conference on Object-Oriented Programing, June 1997.

[8] A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, June 2000.

A Declarative Meta-Programming Approach to Framework Documentation

Tom Tourwé & Tom Mens
{tom.tourwe,tom.mens}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050-Brussel-Belgium

Abstract

The documentation of software artifacts in general, and object-oriented frameworks in particular, has always been problematic. In this paper, we advocate the use of a declarative meta-programming environment to document software artifacts. In particular, we show how a significant and important part of the design of a framework can be adequately and concisely documented in such an environment, and how this allows us to use this documentation in an active way.

1 Introduction

Documentation of software artifacts has been, and appears to remain, a major problem. Documentation is often non-existent, hopelessly out of date and/or inconsistent with the current state of the implementation [3]. Developers, who are responsible for evolving and maintaining those artifacts, are thus heavily discouraged to use the documentation, and are tempted not to read even the most well-crafted documentation [9]. Consequently, this gives rise to serious problems such as code duplication, design inconsistencies, design erosion [11], architectural drift [4], etc..

In particular for framework-based software development, the framework's design is the most important asset to document [6]. It is the design that should be reused by many different applications, and it is the design that should be changed when the framework should evolve. In what follows, we will show how a declarative meta-programming environment can be used to document (part of) the design of a framework, and how it enables us to use this information in an *active way*, rather than passively as is the case now. This active use includes checking the completeness of the documentation, e.g. whether all important parts are included, and its correctness, e.g. whether it is still consistent with the current state of the implementation. As such, we are able to detect whenever this documentation is out of date, and which of its parts are affected.

2 Why Declarative Meta Programming?

We conducted our experiments in the SOUL declarative meta-programming environment [12]. We believe such an environment is extremely well suited for documenting a framework and using its documentation actively, for the following reasons:

The declarative nature of SOUL, and logic programming languages in general, allows us to represent all sorts of knowledge in a straightforward, accurate and concise way [7]. For our purposes, we will use *logic facts* to describe the appropriate design knowledge.

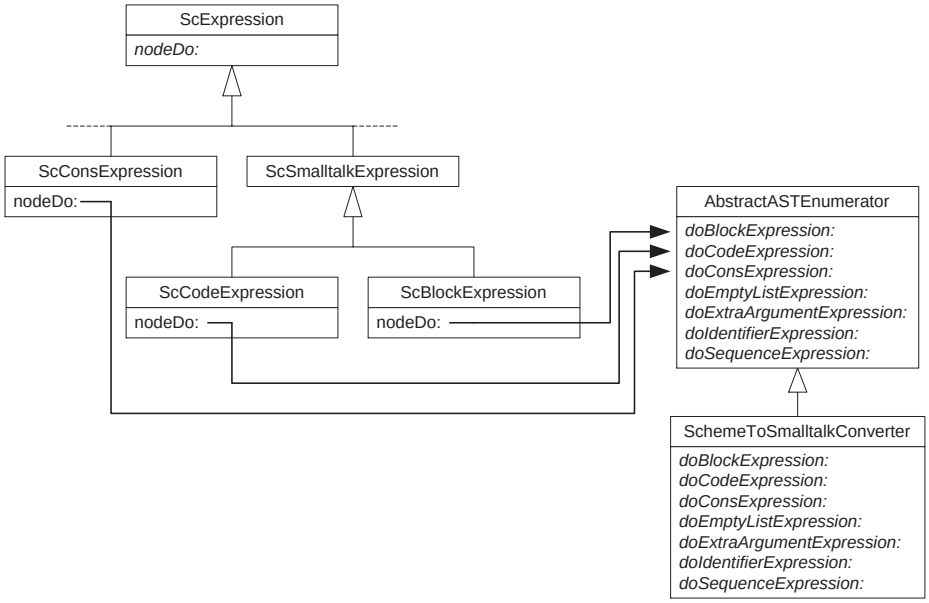

Figure 1: An instance of the *Visitor* design pattern

The powerful reasoning capabilities of logic programming languages are also very useful for our purposes. We can use *logic rules* to derive additional information from existing facts, for example, or to reason about the information represented by these facts. Such reasoning will allow us to check the completeness of the documentation.

The tight integration of SOUL with the standard Smalltalk development environment allows it to consult and reason about the current implementation of the framework. As such, it enables us to write logic rules that access the implementation and check whether the documentation is still correct.

3 Documenting a Framework's Design

Frameworks allow applications to plug in their specific behavior by defining appropriate *hot spots* [8]. Documenting a framework's design thus boils down to documenting the corresponding hot spots, and in particular how these hot spots are expected to be used.

As has already been shown in literature, design patterns are excellent means for documenting the hot spots of a framework [5, 2]. Design patterns define how the hot spots they implement can be used by applications to plug in their specific behavior and expose important information about the particular roles and responsibilities of the classes and methods involved. This is the kind of information that we will explicitly document in our declarative meta-programming environment.

3.1 An Example Design Pattern Instance

Consider the example instance of the *Visitor* design pattern depicted in Figure 1. It shows part of an *expression* hierarchy and its associated *visitor* hierarchy, that are used in a framework for building Scheme interpreters [1, 10]. The *Visitor* design pattern is used in the `ScExpression` hierarchy to allow adding new operations on expressions in a straightforward and flexible way. This simply boils down to adding a new subclass of the `AbstractASTEnumerator` class, and does not require us to change the expression classes.

Moreover, the *Visitor* design pattern defines the particular *participants* that should be present in its implementation, and exposes information about the specific *roles* and *responsibilities* of these participants. For example, it defines *abstractElement* and *abstractVisitor* roles, and requires corresponding class participants to provide an implementation for the *acceptMethod* and *visitMethod* roles respectively.

We can document this particular instance of the *Visitor* design pattern in our declarative meta-programming environment by using logic facts as follows:

```
dpRole(astVisitor,abstractElement,ScExpression).
dpRole(astVisitor,concreteElement,ScConsExpression).
dpRole(astVisitor,concreteElement,ScBlockExpression).
dpRole(astVisitor,concreteElement,ScCodeExpression).
...
dpRole(astVisitor,acceptMethod,nodeDo:).
dpRole(astVisitor,abstractVisitor,AbstractASTEnumerator).
dpRole(astVisitor,concreteVisitor,SchemeToSmalltalkConverter).
dpRole(astVisitor,visitMethod,doConsExpression:).
dpRole(astVisitor,visitMethod,doBlockExpression:).
dpRole(astVisitor,visitMethod,doCodeExpression:).
...
dpRelation(astVisitor,<ScConsExpression,doConsExpression:>).
dpRelation(astVisitor,<ScCodeExpression,doCodeExpression:>).
dpRelation(astVisitor,<ScBlockExpression,doBlockExpression:>).
...
```

The *dpRole* predicate maps roles onto participants. Its first argument is used to identify the particular design pattern instance that is being documented, the second argument denotes the role, and the third argument denotes the class, method or variable playing that role. The *dpRelation* predicate is used to document relations between participants. In this case, it is used to reflect the relation between *concreteElement* and *visitMethod* participants (e.g. the *Visitor* design pattern expects each *concreteElement* participant to define an *acceptMethod* participant that calls a specific *visitMethod* participant).

4 Actively Using the Documentation

Documenting the design of a framework in a meta-programming environment allows us to check the completeness of this documentation and to verify whether it is still up to date with the current implementation. This will be shown in the following sections.

4.1 Checking Completeness

We can easily check whether the documentation of a design pattern is complete, e.g. whether it includes all necessary roles and participants. This requires us to first state which roles a design pattern instance should provide. This is achieved as follows for our *Visitor* design pattern example:

```
requiredRole(visitorDP,abstractElement).
requiredRole(visitorDP,concreteElement).
requiredRole(visitorDP,abstractVisitor).
requiredRole(visitorDP,concreteVisitor).
requiredRole(visitorDP,acceptMethod).
requiredRole(visitorDP,visitMethod).
```

Based on this information, we use a logic rule that consults the documentation of the design pattern instance to see if it effectively includes a description for every required role:

```
checkPatternInstance(?pattern, ?instance, ?absentRoles) if
     findall(?role,
                and(requiredRole(?pattern,?role),
                       not(dpRole(?instance,?role,?))),
                ?absentRoles)
```

As a side effect, this rule returns the list of roles that is not included in the design pattern instance documentation.

Conversely, for some specific participants, we can check whether they are included in the documentation, as they should. The following rule, for instance, checks whether all concrete subclasses of an *abstractVisitor* class participant are registered as *concreteVisitor* participants:

```
checkPatternInstance(?, ?instance, ?absentParticipants) if
     dpRole(?instance,abstractVisitor,?abstractVisitor),
     findall(?role,
                and(hierarchy(?abstractVisitor,?class),
                       concreteClass(?class)
                       not(dpRole(?instance,concreteVisitor,?class))),
                ?absentParticipants)
```

The *hierarchy* predicate returns all (possibly indirect) subclasses of the *abstractVisitor* class participant, while the *concreteClass* predicate checks whether a class is indeed a concrete class.

4.2 Checking Consistency

We are also able to check whether the documentation is still consistent with the current implementation of the framework. This is particularly important given the fact that framework evolve over time.

We can achieve such verification thanks to the fact that design patterns impose constraints upon an implementation, and that we can represent these constraints explicitly in SOUL. For instance, one constraint of the *Visitor* design pattern is that it requires each *concreteElement* participant to be a (possibly indirect) subclass of the *abstractElement* participant. We can express this constraint in a logic rule that consults both the documentation and the implementation and checks whether they are consistent, as follows:

```
patternConstraint(visitorDP,?instance,incorrectCE(?violators)) if
[1] dpRole(?instance,abstractElement,?abstractElement),
     findall(?concreteElement,
[2]             and(dpRole(?instance,concreteElement,?concreteElement),
[3]                    not(hierarchy(?abstractElement,?concreteElement))),
                ?violators)
```

The logic predicates at line 1 and 2 consult the documentation of the design pattern instance, whereas the *hierarchy* predicate consults the implementation to see if the correct inheritance relation holds between the two classes. Classes that do not adhere to the above rule are reported as *incorrect concrete element* participants.

The above example only shows one constraint for the *Visitor* design pattern. Others can be defined in a similar way. Furthermore, similar constraints can be implemented in a similar way for other design patterns as well.

5 Conclusion

In this paper, we have shown how declarative meta-programming can be used for documenting a framework's design. Thanks to the declarative nature of SOUL, we achieved this in a straightforward, accurate and concise way. Moreover, thanks to SOUL's powerful reasoning and meta-programming capabilities, we were able to use this documentation actively to check whether it is both complete and correct with respect to the current implementation.

We strongly believe this is an important first step towards better and more active documentation for frameworks and software artifacts in general. The ideas presented in this paper actually form part of a more general approach to document and reason about a framework's implementation, its instantiation and evolution at a high-level. We refer the interested reader to [10].

References

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Kent Beck and Ralph Johnson. Patterns Generate Architectures. In *Proceedings of the European Conference on Object-Oriented Programming*, 1994.

[3] Greg Butler and Pierre Dénommée. *Documenting Frameworks to Assist Application Developers*, chapter 7. John Wiley and Sons, 1999.

[4] C. B. Jaktman, J. Leaney, and M. Liu. Structural Analysis of the Software Architecture – a Maintenance Assesment Case Study. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*. Kluwer Academic, 1999.

[5] Ralph Johnson. Documenting Frameworks Using Patterns. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1992.

[6] Ralph Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1988.

[7] G. F. Luber and W. A. Stubblefield. *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 1998.

[8] Wolfgang Pree. Essential Framework Design Patterns. *Object Magazine*, 1997.

[9] Mark Rettig. Nobody Reads Documentation. *Communications of the ACM*, 34(7):19–24, 1991.

[10] Tom Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.

[11] Jilles van Gurp and Jan Bosch. Design Erosion: Problems & Causes. *Journal of Systems & Software*, 61(2):105–119, 2001.

[12] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.

Automatic Normalisation via Metamodelling

D.H.Akehurst, B.Bordbar, P.J.Rodgers, N.T.G.Dalgliesh

University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF
{ D.H.Akehurst, B.Bordbar, P.J.Rodgers, N.T.G.Dalgliesh }@ukc.ac.uk

The process of normalisation has long been accepted as a crucial part of the design for good database systems. By using a declarative approach to the specification of the normalisation rules and a precisely defined transformation, over a meta-model of a database system design language, we can automate the normalisation process. A tool supporting the normalisation of database system designs can subsequently be developed providing an invaluable aid to the software system designer.

Introduction

The Unified Modelling Language (UML [1]) is becoming the industry standard for expressing database schemas [2]. A fundamental part of schema design is the process of Normalisation. This process, which is well documented in the literature [3] [4], is designed to guarantee data integrity in the model by using formal techniques to eliminate redundancy and the possibility of data update anomalies. The process consists of the stepwise application of a sequence of transformations on the schema in order to meet defined constraints, each of which describes a *normal form* – first (1NF), second (2NF), third (3NF) and Boyce-Codd (BCNF) normal forms.

All normal form definitions make use of *functional dependencies* between attributes. A functional dependency is a pair consisting of a set of attributes (the determinant) whose values can be used to uniquely identify the values of another set of attributes (the dependant). For example, a property identification number can be used to identify a unique property address and the rent charged for that property.

Through using UML as the schema definition language we gain access to the Object Constraint Language (OCL [1]), a declarative constraint, navigation and logical expression language. This is used to construct expressions that encode functional dependencies between attributes and the above four normal form definitions on classes at a meta-level. These OCL expressions provide a concise, easily understandable and generic mechanism for reasoning about the specific normal form of any schema instantiated from the meta-model.

The transition steps from one normal form to the next can be described as a series of transformations on the model. The transformations take the form of replacing each class from the schema that is in the lower normal form with two or more new classes (and linking associations) that conform to the higher normal form. This transformation can also be encoded as a transformation over the UML meta-model, and thus be used as a generic transformation, applicable to any schema definition.

The combination of these two meta-level specifications is used to construct a schema transformation tool that takes as input a UML definition of a database system schema along with definitions of the functional dependencies, and produces as output a new UML schema that is conformant to one or more of the defined normal forms.

The rest of this paper describes: the UML profile necessary for modelling database system schemas; the meta-level encoding of the four normal forms in OCL; and finally the tool built to automate this process, including the meta-transformation used to convert schemas between normal forms. The paper concludes with an over view of the achievements and a look towards future work.

UML for Data Modelling

A number of suggestions have been made for a data modelling profile in UML [5] [6]. These profiles are well suited for the specification of models of data, both in relational and object-based databases. However, our objective, to support the normalization process, requires a more precise semantics including a set of well-formedness rules, neither of which is defined in the referenced work.

We define a set of stereotypes and tagged value that extend the UML meta-model to include the relational data modelling concepts, such as Keys, Functional Dependencies, and Table Definitions. Additionally, well-formedness rules are defined that specify the logical connections between the concepts. Subsequently we enhance the profile by formally defining the semantics of the concepts in terms of the 'instance' metamodel elements. Finally we add the normalization operations that can be used to verify if a class is in one or other of the normal forms.

The semantics and well-formedness rules are not of direct relevance to this paper, hence we simply include a summary of our UML profile, which defines the following stereotypes and tagged values:

Stereotype	UML meta-model baseClass
«Schema»	Package
«TableDefiniton»	Classifier
«Field»	Attribute
«FunctionalDependency»	Dependancy

Tag	Applies To
PK : Boolean	«Field»
CK : Set(Integer)	«Field»
FK : (Seq(«Field»), Seq(«Field»))	AssociationEnd

For examples of each of these extensions see Figure 2 (below), which contains a class diagram specifying part of a schema definition.

Schema

A schema is the collection of Table Definitions and inter-relationships that form the specification of the database system. We stereotype the UML Package element in order to identify that a particular package defines a database system. «Schema» packages contain «TableDefinition» classes and relationships between them.

Table Definition

A data class defines the type of its instances; it can therefore be taken to define the structure of a corresponding table where attributes and their types define column names and their domains. We introduce a stereotype «TableDefinition» that extends the concept of Classifier (from the UML meta-model).

Functional Dependency

To express a functional dependency we extend the 'Dependency' concept, introducing the stereotype «FunctionalDependency». The well-formedness rules constrain this to a dependency between attributes.

The notation for depicting a functional dependency between sets of attributes follows the 'dashed arrow' syntax specified for the dependency relationship in the UML standard (see section 2). However, avoid excess clutter on a diagram they can also be expressed textually as a pair of sets linked by an arrow '→', in line with the notation used in relational database modelling.

Candidate Key

We use a tagged value named 'CK' to indicate that an attribute is part of a candidate key for a Table Definition. An attribute may form part of more than one candidate key, hence we specify the data value of the tagged value to be a set of integers. Each integer indicates that the attribute is part of a different key.

The UML standard does not give any indication of syntax for the definition of tagged values on attribute stereotypes; we conform to the notation for tagged values defined on Classes, by adding the name and value within braces (e.g. '{CK = 1,2 }'), alongside the relevant attribute.

Primary Key

It is possible for a table definition to have more than one candidate key. In such a case the relational model has historically required that exactly one of those keys to be chosen as the *primary key*. To indicate the primary key we define an addition tagged value with a Boolean data type named 'PK'.

To depict that an attribute forms part of the primary key the name 'PK' is added to the defined tagged values for the attribute; e.g. '{PK, CK = 2,3 }'. (Boolean tagged values do not require an explicit value if being set to true.)

Foreign Key

A foreign Key is a set of attributes that map to the primary key of an associated Table Definition. Associations indicate the relationships between Table Definitions. Standard UML semantics assume the presence of a unique object identifier, and this is used to semantically distinguish between different objects. However, in Relational database semantics, the primary key forms this distinction. Thus, for an Association instance (Link) to be navigated it is necessary to know which attributes contain the foreign key values that identify the 'object' at the other end of an association. We indicate this by using a tagged value 'FK' on an AssociationEnd. This tag defines the attribute names that form the foreign key, e.g. {FK=p#,c#}.

Meta Encoding of the Normal Forms

The following operations are defined for the «TableDef» stereotype. They all return a Boolean value indicating whether or not the table is in one of the defined Normal Forms. For a Schema to be in one of the normal forms, all of the contained Table definitions must be in that normal form. It is necessary for a particular Table definition to contain the specification of Primary Keys, Functional Dependencies, and Candidate Keys in order for these operations to be successful.

is1NF() : Boolean

UML class diagrams with PKs are in 1NF. We define an OCL expression that fails if there are no PKs defined, i.e. a table is not in 1NF if there are no PKs defined.

```
result:
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
attributes->exists(a | a.taggedValue->contains(tg |
                              tg.name = 'PK' and tg.dataValue=true) );
```

is2NF() : Boolean

A table is in 2NF if it is in 1NF, plus every non-key attribute is functionally dependent on the full key.

```
result: is1NF and
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
Let dependencies = attributes.clientDependency->select(d |
                           d.stereotype.name=`«FunctionalDependency»') in
Let pk = attributes->select(a|a.taggedValue->contains(tg |
                              tg.name = 'PK' and tg.dataValue=true)) in
Let npks = attributes - pk in
  npks->forAll( na |
         dependencies->exists( d | d.supplier->contains(na) and d.client = pk )
```

is3NF() : Boolean

Class must be in 2NF, plus every non-key attribute should be functionally dependent on the full key and nothing but the full key

```
result: is2NF and
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
Let dependencies = attributes.clientDependency->select(d |
                           d.stereotype.name=`«FunctionalDependency»') in
Let pk = attributes->select(a|a.taggedValue->contains(tg |
                              tg.name = 'PK' and tg.dataValue=true)) in
Let npks = attributes - pk in
  npks->forAll( na |
     dependencies->select(d|d.supplier.contains(na))->forAll(d|d.client = pk) )
```

This constraint says, that for all dependencies containing a non-PK attribute in the client part, the supplier part of that dependency must be the PK.

isBCNF() : Boolean

A relation is in Boyce-Codd Normal Form (BCNF) if every determinant is a candidate key.

```
result:
Let attributes = self.feature->select(oclIsKindOf(Attribute)) in
Let dependencies = attributes.clientDependency->select(d |
                           d.stereotype.name=`«FunctionalDependency»') in
Let cks = attributes.taggedValue->select(name=`CK').dataValue->
           flatten->collect( i |
              attributes->select( a | a.taggedValue->contains(tg |
                 tg.name=`CK' and tg.dataValue=i) ) in
  dependencies->forAll(d | cks->contains(d.client) )
```

Tool Support

Work at UKC is almost complete regarding the production of a tool (illustrated in Figure 1) to support the automatic normalisation process described in this paper. The tool accepts an XMI encoding of the UML schema, and a (possibly separate) XMI encoding of the functional dependencies. These are converted into a combined java object model of the schema definition using IBM's XMI framework [7] and additional libraries developed at UKC [8].

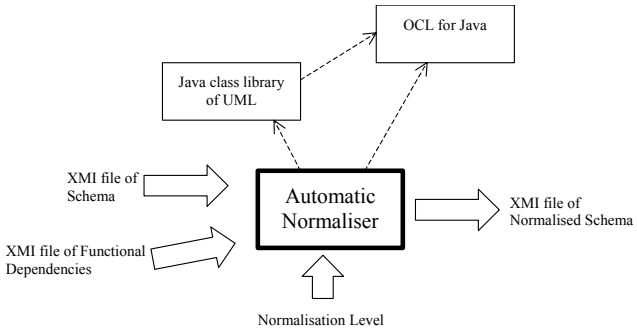

Figure 1

The java object model is based on a set of java classes that encode the standard UML meta model. This set of classes has been automatically generated from an XMI encoding of the UML meta-model (using a simpler form of the generator to boot strap the process).

The generated java classes are built on an additional library that implements the OCL types and functions in such a way that OCL expressions can be evaluated over a java object model. The normalisation level of the java representation of schema can thus be deduced by evaluating the OCL expressions (defined above) using the OCL library.

A transformation algorithm is applied to the combined model, generating a new output model defining a schema that conforms to the required normal form. The core part of the transformation algorithm is the replacement of one class with two others as described below:

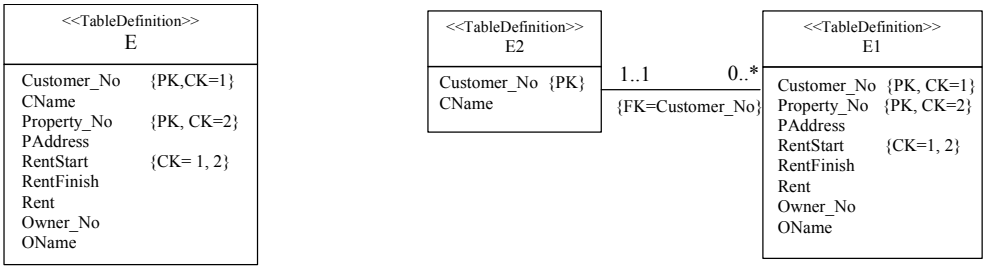

Figure 2

For a class ('E' from Figure 2) and a functional dependency A→B (e.g. {Customer_No} → {Cname}) we can perform a rewrite that replaces the original class with two new ones ('E1' and 'E2' from Figure 2). The re-writing rule for a class E into classes E1 and E2 is described as follows:

- Class E2 will contain the set of elements A (the determinant of the dependency we are decomposing) and any elements wholly dependent on A, which trivially includes B. The primary key of this class is the set of attributes A.
- Class E1 contains the elements not in class E2, except for elements of A. The set A remains in both classes, representing the association between them: as the foreign key of E1 linking to the primary key of E2. The primary key of class E1 is the same primary key of the original class E.

This algorithm is repeatedly applied to the classes and functional dependencies of the model, in order to transform the schema, as a whole, into the required normal form.

Conclusion

The work described in this paper has shown a method of automating the process of normalisation as defined within the context of database systems. The automation is achieved through the use of two declarative specifications over the UML meta-model.

1. The use of OCL declarations over a data-modelling profile of UML to encode the definitions of the four normal forms (1NF, 2NF, 3NF and BCNF).
2. The definition of a graph rewriting rule to specify the transformation steps required for converting a data model from one normal form to a higher normal form.

These specifications have been used to construct a tool that implements the transformation thus automating the normalisation process.

We plan to formalise the transformation process as a specification over the UML meta-model using graph transformations [9] or some other appropriate technique. We also plan to extend the tool support to include automatic generation of SQL Schema definitions from the UML models.

References

[1] OMG, "The Unified Modeling Language Version 1.4," Object Management Group formal/01-09-67, 2001.
[2] E. J. Naiburg and R. A. Maksimchuk, *UML for Database Design*: Addison Wesley, ISBN 0-201-72163-5, 2001.
[3] J. Paredaens, P. De-Bra, M. Gyssens, and D. Van-Gucht, *The Structure of the Relational Database Model*: Springer-Verlag, ISBN 3-540-13714-9, 1989.
[4] C. J. Date, *An Introduction to Database Systems (Introduction to Database Systems 7th Ed)*: Addison Wesley Publishing Company, ISBN 0201385902, 1999.
[5] S. Ambler, "Persistence Modeling in the UML," 1999, http://www.sdmagazine.com
[6] Rational, "The UML and Data modelling," 2000, www.rational.com
[7] IBM, "alphaWorks XMI Framework," 2001, http://www.alphaworks.ibm.com/tech/xmiframework
[8] D. H. Akehurst, "OCL for Java," 2002, http://www.cs.ukc.ac.uk/people/staff/dha/xInterests/UMLandOCL/
[9] *Handbook of graph grammars and computing by graph transformation*, Vol. 1, Foundations, edited by G. Rozenberg. World Scientific, Singapore, 1997.

Realizing Aspects by Transforming for Events

Robert E. Filman[1] and Klaus Havelund[2]

[1] RIACS
NASA Ames Research Center, MS/269-2
Moffett Field, CA 94035
rfilman@mail.arc.nasa.gov
[2] Kestrel Technology
NASA Ames Research Center, MS/269-2
Moffett Field, CA 94035
havelund@email.arc.nasa.gov

Abstract. We explore the extent to which concerns can be separated in programs by program transformation with respect to the events required by these concerns. We describe our early work on developing a system to perform event-driven transformation and discuss possible applications of this approach.

1 Aspect-Oriented Programming

Programming is about realizing a set of requirements in an operational software system. One has a (perhaps changing) set of properties desired of a system, and builds and evolves that system to achieve those properties. Software engineering is the accumulated set of processes, methodologies, and tools to ease that evolutionary process, including techniques for figuring out what it is that we want to build and mechanisms for producing higher-quality systems.

A recurrent theme of software engineering is that of "separation and localization of concerns." That is, we have "concerns" in building a software system. These concerns range from high-level ilities like reliability and security to low-level issues like caching and synchronization. Our environment should provide us with the linguistic structures to group together just the elements for a particular concern, while nevertheless yielding an efficient operational system. This allows us to concentrate expertise on that concern in one particular place, easing system development and maintenance.

Conventional programming languages (e.g., procedural, imperative and functional languages) provide only a few facilities for separating concerns. The guiding theme of these languages is functional decomposition. One determines what the elements of the domain are and what behaviors they need to have, and writes code to implement these methods. The writer of this code must take account any other requirements beyond pure functionality in the actual code that isn't somehow otherwise supported in the environment. The insight of Object-Oriented Programming was recognizing the leverage of localizing concerns centered on functionality of the elements of the domain (in objects), indexing behavior with

respect to these objects (methods on objects) and providing mechanisms for acquiring default behavior and values (inheritance).

Object-oriented decomposition is often good for the "dominant" functional concern of the system, but leaves other concerns unsupported. This is especially true for those concerns that require coherent behavior across many different functionalities. The best conventional programming can offer is to concentrate the code of other concerns in another function or object, and demand programmers explicitly invoke that code when appropriate. But spreading out the responsibility for invoking the code for multiple concerns to all programmers produces a more brittle system. Each programmer who has to do something right is one more place that a mistake can be made; each spot where something needs to be done is a potential maintenance mishap. Additionally, there may be execution costs in control transfer. Some separate concerns may require so much local context that they may not even be expressible as separate subprograms.

Object-Orientation hasn't given us is any leverage on the problem of cross-cutting requirements and behaviors—elements that require matching code in many places in a system, but which are not neatly packagable in the standard object decompositions. Aspect-Oriented Programming (AOP) (and, more generally, Aspect-Oriented Software Development (AOSD)) is an emerging technology for creating programming systems which provide a "single locus" for expressing such cross-cutting behavior while nevertheless creating systems that actually execute efficiently. The general theme of AOSD is to let programmers express the behavior for each concerns in its own element. Such a system must also include some directions for how the different concerns are to be knitted together into a working system (for example, which each separate concern applies) and a mechanism for actually producing a working system from these elements. For example, most AOP systems given one a way of saying, "High security is achieved by doing X. Reliability is achieved by doing Y. I want high security in the following places in the code, and reliability on these operations." The AOP system then produces an object that invokes the high security and reliability codes appropriately.

2 AOP and Events

Elsewhere, we have argued that the programmatic essence of Aspect-Oriented Programming is making quantified programmatic assertions over programs that otherwise are not annotated to receive these assertions [1–3]. That is, in an AOP system, one wants to be able to say things of the form, "In this program, when the following is true, do the following," without having to go around marking the places that need the desired modification. Most naturally, most of the kinds of quantified statements that programmers want to make are about behavior that is to take place when certain conditions are realized in the executing program. Consider some AOP applications:

Synchronization in distributed systems [4, 5] Code to check the synchronization condition ought to run before and after synchronization operations.

This code needs internal state (for example, lock state and a perhaps a queue of waiting processes.)

Buffer manipulation in operating systems kernels [6] In operating systems code, prefetching of pages is to be executed when a page fault event occurs within the run-time execution context of a prefetch advisory. Entering and leaving such contexts are also events.

Distributed middleware [7–9] Aspects can run on inter-object communication in distributed systems to check for consistent configurations, automate testing, and provide a great variety of other dynamically configurable behavior.

Distributed quality-of-service [10, 11] By intercepting service invocation events, aspects can be used to regulate quality of service in concurrent systems.

Collaboration and design [12, 13] By interceding at service invocation and repository entry, aspects can be used to enforce access control, synchronization, persistence and resource management in collaborative systems.

E-commerce [14, 15] Douence et. al provide an example of using complex histories of client events to determine e-commerce prices. Truyen et. al argue that appropriate aspect behavior is controlled by a complex context set up by a sequence of user actions.

Replication [16–18] Replication through aspects is accomplished by taking each action that changes state and propagating it to the replicants.

Debugging [3] AOP techniques can be used to create the trace of events to track program execution or to insert the switching commands to force (concurrent) programs to explore multiple program paths.

Program instrumentation [19] AOP techniques can be used to insert logging information on system performance at interesting junctures in program execution.

Most conventional AOP systems rely primarily on wrapping function calls with aspect behavior. However, these examples illustrate the need to be able to respond to sequences of events, actions at the individual statement level, and properties of the state of the modeled system.

So what is an event? Ultimately, we want to be able to quantify over anything that changes the data or program counter state of the abstract machine executing a given program. Unfortunately, the abstract interpreter is not completely accessible at the programming level—it is neither fixed by the language definition nor do all its activities (for example, thread switching and garbage collection) have any visible realization in the program text. Similarly, an optimizing compiler may rearrange or elide an "obvious" sequence of expected events. And finally, the data state of the abstract interpreter (including, as it does, all of memory) can be a grand and awkward thing to manipulate.

Nevertheless, much of a program is accessible—we do, after all, have the program text (or the byte code), and can manipulate that code to our heart's content. We may not be able to capture everything that goes on in a particular interpretive environment, we can get close enough for many practical purposes. The strategy we adopt is to argue that most dynamic events, while not necessarily local to a particular spot in the source code, are nevertheless tied to places in

the source code. Table 1 illustrates some primitive events and their associated code loci.

Users are likely to want to express more than just primitive events. The language of events will also want to describe relationships among events, such as that one event occurred before another, that a set of events match some particular predicate, that an event occurred within a particular timeframe, or that no event matching a particular predicate occurred. This suggests that the event language will need (1) abstract temporal relationships, such as "before" and "after," (2) abstract temporal quantifiers, such as "always" and "never", (3) concrete temporal relationships referring to clock time, (4) cardinality relationships on the number times some event has occurred, and (5) aggregation relationships for describing sets of events.

Event	**Syntactic locus**
Accessing the value of a variable or field	References to that variable
Modifying the value of a variable or field	Assignments to that variable
Invoking a subprogram	Subprogram calls
Cycling through a loop	Loop statements
Branching on a conditional	The conditional statement
Initializing an instance	The constructors for that object
Throwing an exception	Throw statements
Catching an exception	Catch statements
Waiting on a lock	Wait and synchronize statements
Resuming after a lock wait	Other's notify and end of synchronizations
Testing a predicate on several fields	Every modification of any of those fields
Changing a value on the path to another	Control and data flow analysis over statements (slices)
Swapping the running thread	Not reliably accessible, but atomization may be possible
Being below on the stack	Subprogram calls
Freeing storage	Not reliably accessible, but can try using built-in primitives
Throwing an error	Not reliably accessible; could happen anywhere

Table 1. Table 1: Events and event loci

We are currently working on a system where a set of event-action pairs, along with a program, would be presented to a compiler. Each event action pair would include a sentence describing the interesting event in the event language and an action to be executed when that event is realized. Said actions would be programs, and would be parameterized with respect to the elements of the matching events. Examples of such assertions are:

- On every call to method foo in a class that implements the interface B, replace the second parameter of the call to foo with the result of applying method f to that parameter.
- Whenever the value of x+y in any object of class A ever exceeds 5, print a message to the log and reset x to 0.

- If a call to method foo occurs within (some level down on the stack) method baz but without an intervening call to method mumble, omit the call to method gorp in the body of foo.
- Every call to foo must be followed by a call to baz without an intervening call to mumble.

These examples are in natural language. Of course, any actual system will employ something formal.

Clearly, a sufficiently "meta" interpretation mechanism would give us access to many interesting events in the interpreter, enabling a more direct implementation of these ideas. It has often been observed that meta-interpretative and reflective systems can be used to build AOP systems [20]. However, meta-interpreters have traditionally exhibited poor performance. We are looking for implementation strategies where the cost of event recognition is only paid when event recognition is used. This suggests a compiler that would transform programs on the basis of event-action assertions. Such a compiler would work with an extended abstract syntax tree representation of a program. It would map each predicate of the event language into the program locations that could affect the semantics of that event. Such a mapping requires not only abstract syntax tree generation (parsing) and symbol resolution, but also developing primitives with respect to the control and data flow of the program, determining the visibility and lifetimes of symbols, and analyzing the atomicity of actions with respect to multiple threads.

Java compiles into an intermediate form (Java byte codes). In dealing with Java, there is also the choice as to whether to process with respect to the source code or the byte code. Each has its advantages and disadvantages. Byte codes are more real: many of the issues of interest (actual access to variables, even the power consumption of instructions) are revealed precisely at the byte code level. Working with byte codes allows one to modify classes for which one hasn't the source code, including the Java language packages themselves. (JOIE [21] and Jmangler [22] are examples of an AOP systems that perform transformations at the byte code level.) On the other hand, source code is more naturally understandable, allows writing transformations at the human level, and eliminates the need for understanding the JVM and the actions of the compiler. (De Volder's Prolog-based meta-programming system is an example of source-level transformation for AOP [23, 24].) We find the complexity arguments appealing. Thus, our implementation plan is to work at the source code level.

3 Related Work

De Volder and his co-workers [23, 24] have argued for doing AOP by program transformation, using a Prolog-based system working on the text of Java programs. We want to extend those ideas to program semantics, combining both the textual locus of dynamic events and transformations requiring complex analysis of the source code.

At the 1998 ECOOP AOP workshop, Fradet and Südholt [25] argued that certain classes of aspects could be expressed as static program transformations. They expanded this argument at the 1999 ECOOP AOP workshop to one of checking for robustness—non-localized, dynamic properties of a system's state [26]. Colcombet and Fradet realized an implementation of these ideas in [27], applying both syntactic and semantic transformations to enforce desired properties on programs. In that system, the user can specify a desired property of a program as a regular expression on syntactically identified points in the program, and the program is transformed into one that raises an exception when the property is violated. Other transformational systems include, Ku a notational attempt at formalizing transformation [28], and Schonger et al's proposal to express abstract syntax trees in XML and use XML transformation tools for tree manipulation [29].

Nelson et al. identify three concern-level foundational composition operators: correspondence, behavioral semantics and binding [30]. Correspondence involves identifying names in different entities that are "the same"—for data items, things that should share storage; for functions, functional fragments that need to be assembled into a whole. Behavioral semantics describe how the functional fragments are assembled. Binding is the usual issue of the statics and dynamics of system construction and change. They discuss alternative formal techniques for establishing properties of composed systems within this basis.

Masuhara et. al present a semantics-based approach to compiling AOP systems. They introduce the notion of "join point shadows"—the places in the text where the a particular aspect needs to be woven [31].

Walker and Murphy argue for events as appropriate "join points" for AOP, and that the events exposed by AspectJ are inadequate [32].

4 Concluding remarks

We have suggested that an interesting way to implement AOP systems is by describing the events that are to trigger aspect behavior, and transforming an existing program with respect to these events. We note that we've been considering implementation environments, not software engineering. An underlying implementation does not imply anything about the "right" organization of "separate concerns" to present to a user. In particular, we have been completely agnostic about the appropriate structure for the actions of action-event pairs. It may be the case that unqualified use of an event language with raw action code snippets is a software engineering wonder, but we doubt it. On the other hand, we believe that such transformational system would be an excellent environment for experimenting with and building systems for AOP. In some sense, these ideas can be viewed as a domain-specific language for developing aspect-oriented languages.

References

1. Filman, R.: What is aspect-oriented programming, revisited. [33]

2. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns (OOPSLA 2000). (2000)
3. Filman, R.E., Havelund, K.: Source-code instrumentation and quantification of events. [34] 45–49
4. Holmes, D., Noble, J., Potter, J.: Aspects of synchronization. In: Workshop on Aspect Oriented Programming (ECOOP 1997). (1997)
5. Netinant, P., Elrad, T., Fayad, M.E.: A layered approach to building open aspect-oriented systems: A framework for the design of on-demand system demodularization. Comm. ACM **44** (2001) 83–85
6. Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., Ong, J.S.: Structuring operating system aspects: Using AOP to improve OS structure modularity. Comm. ACM **44** (2001) 79–82
7. Filman, R.E., Barrett, S., Lee, D.D., Linden, T.: Inserting ilities by controlling communications. Comm. ACM **45** (2002) 116–122
8. Hunleth, F., Cytron, R., Gill, C.: Building customizable middleware using aspect oriented programming. [35]
9. Jørgensen, B.N., Truyen, E., Matthijs, F., Joosen, W.: Customization of object request brokers by application specific policies. In: Proc. Middleware'2000. (2000)
10. Becker, C.: Quality of service and O.O. oriented middleware multiple concerns and their separation. [36] 117–126
11. Zinky, J., Shapiro, R., Loyall, J., Pal, P., Atighetchi, M.: Separation of concerns for reuse of systemic adaptation in quo 3.0. [33]
12. Filman, R.E.: A software architecture for intelligent synthesis environments. In: Proc. 2001 IEEE Aerospace Conference. (2001) 2879–2888
13. Pinto, M., Amor, M., Fuentes, L., Troya, J.: Collaborative virtual environment development: An aspect-oriented approach. [36] 97–102
14. Douence, R., Motelet, O., Südholt, M.: Sophisticated crosscuts for e-commerce. [33]
15. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jørgensen, B.N.: Customization of on-line services with simultaneous client-specific views. [33]
16. Antunes, M., Miranda, H., Silva, A.R., Rodrigues, L., Martins, J.: Separating replication from distributed communication: Problems and solutions. [36] 103–110
17. Filman, R.E., Lee, D.D.: Redirecting by injector. [36] 141–146
18. Herrero, J.L., Sánchez, F., Toro, M.: Fault tolerance AOP approach. In: Workshop on Aspect-Oriented Programming and Separation of Concerns (Lancaster). (2001)
19. Deters, M., Cytron, R.K.: Introduction of program instrumentation using aspects. [35]
20. Sullivan, G.T.: Aspect-oriented programming using reflection and meta-object protocols. Comm. ACM **44** (2001) 95–97
21. Cohen, G.A.: Recombing concerns: Experience with transformation. In: Workshop on Multi-Dimensional Separation of Concerns (OOPSLA 1999). (1999)
22. Kniesel, G., Costanza, P., Austermann, M.: JMangler—a framework for load-time transformation of Java class files. In: First IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM 2001). (2001)
23. De Volder, K., Brichau, J., Mens, K., D'Hondt, T.: Logic meta-programming, a framework for domain-specific aspect programming languages. (http://www.cs.ubc.ca/~kdvolder/binaries/cacm-aop-paper.pdf)
24. Volder, K.D., D'Hondt, T.: Aspect-oriented logic meta programming. In Cointe, P., ed.: Meta-Level Architectures and Reflection, 2nd Int'l Conf. Reflection. Volume 1616 of LNCS., Springer Verlag (1999) 250–272

25. Fradet, P., Südholt, M.: AOP: Towards a generic framework using program transformation and analysis. In: Workshop on Aspect Oriented Programming (ECOOP 1998). (1998)
26. Fradet, P., Südholt, M.: An aspect language for robust programming. In: Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999). (1999)
27. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Proc. 27th ACM Symp. on Principles of Programming Languages. (2000) 54–66
28. Skipper, M.: A model of composition oriented programming. In: Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000). (2000)
29. Schonger, S., Pulvermueller, E., Sarstedt, S.: Aspect oriented programming and component weaving: Using XML representations of abstract syntax trees. In: Workshop Aspektorientierte Softwareentwicklung (Bonn), Institut für Informatik III, Universität Bonn (2002)
30. Nelson, T., Cowan, D., Alencar, P.: Supporting formal verification of crosscutting concerns. In Yonezawa, A., Matsuoka, S., eds.: Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192, Springer-Verlag (2001) 153–169
31. Masuhara, H., Kiczales, G., Dutchyn, C.: Compilation semantics of aspect-oriented programs. [34] 17–26
32. Walker, R.J., Murphy, G.C.: Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In: Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001). (2001)
33. Workshop on Advanced Separation of Concerns (ECOOP 2001). In: Workshop on Advanced Separation of Concerns (ECOOP 2001). (2001)
34. FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002). In: FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002). (2002)
35. Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001). In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001). (2001)
36. Proc. Int'l Workshop on Distributed Dynamic Multiservice Architectures (ICDCS-2001), Vol. 2. In: Proc. Int'l Workshop on Distributed Dynamic Multiservice Architectures (ICDCS-2001), Vol. 2. (2001)

SML Prototypes from Z Specifications

Greg Michaelson

Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, EH14 4AS, UK
greg@cee.hw.ac.uk

Abstract. The wider uptake of Declarative Meta-Programming may be constrained by a perceived separation of formal notations and declarative techniques from practical system construction. A pedagogic approach to demonstrating the relevance of formal and declarative techniques, through the prototyping Z of specifications in Standard ML, is discussed.

1 Introduction

In principle, Declarative Meta-Programming (DMP) offers a promising realisation of formal techniques for software development through tools written in declarative languages. DMP is based on manipulation of program code, often specified as compositional abstract syntax tree transformations in some formal notation. Such notations tend to have a close correspondence to declarative languages; hardly surprising given that formalisms for describing language semantics have common roots with declarative languages in number theoretic predicate calculus, set theory, recursive function theory and λ calculus. Thus, declarative languages are a good first choice for practical experimentation with DMP.

However, the wider uptake of DMP may be compromised by the unwarranted aura of academic obscurity and mathematicity that attends its foundations. In particular, formal approaches and declarative programming are often taught in isolated, specialist modules in Computer Science programmes. Thus, students see little relevance to mainstream software engineering and, as Computing professionals, are unlikely to promote them once in employment.

The origins of these techniques' isolation may lie in the Platonist views of many of their strongest supporters. Such views gained currency in the 1970s, in debates about the relative merits of operational and denotational semantics (DS), summarised in [Mic93]. Members of the Oxford Programming Research Group who developed DS, argued for its primacy as static descriptions of ideal abstract entities rather than as potentially executable specifications of languages. In particular, they asserted that an operational reading of a DS imputed unnecessary, non-unique, implementation details [SS71,Sco72,Sto77].

More recently, Hayes and Jones [HJ89] argued against the executability of specifications in general. Like the DS proponents, they asserted that specification notations encompass infinite domains and non-computable constructs, and underdetermine equivalent implementations. They also said that executable

notations require implementation properties that are irrelevant from a formal perspective.

Ironically, people working with formal specifications often wish to explore empirically their properties. Thus, specifications may be "animated" using symbolic techniques like abstract interpretation. If concrete values are substituted for symbols, and animation terminates, then the final result might be interpreted as an output from the execution of an equivalent program.

In contrast, there has been considerable research into refinement techniques for deriving software from specifications. However, the target has usually been unimplemented Djikstra/Hoare/Gries guarded command notations for which proof techniques are easily elaborated [Kal90,Mor90]. Furthermore, there is a dearth of tools to aid refinement to standard languages.

The significant exception to these gloomy observations has been the development of Abrial's B-Method [Abr96], which integrates specification, proof, refinement and implementation through the unifying concept of abstract machines. Refinement is based on successive steps to replace abstract with more concrete constructs. Each step must be justified through a linking invariant, which relates the initial and refined states. The invariant also provides assurances about the properties of concrete constructs.

2 Prototyping with specifications

Since 1992, we have been teaching a module on formal specification in Z [Spi89] to 2nd year BSc Computer Science students. In a recent attempt to increase relevance, we sought to develop a linked module where the students would construct programs from Z specifications.

An initial candidate was the B-Method, which certainly offers good tool support for proof and animation. However, the B notation is far closer to Z than to a standard programming language, the use of which we deemed important for emphasising the significance of formal specification for practical software development. Thus, we decided instead to trade B's rigour for a more informal approach based on the direct translation of Z to light weight prototypes in a standard language. We also thought that this would enable relatively quick experimentation with the implementation implications of specifications,

In choosing a target language, we sought a close correspondence to Z. Z, based on set theoretic predicate calculus, has the Church-Rosser property of evaluation order independence, is strongly typed and enables parametric polymorphism through generic schemas. This suggested the use of a declarative language as the target.

We chose Standard ML (SML) [MTH97], a strict, strongly type, parametric polymorphic language. SML is technically an imperative language with a pure functional subset. We thought that SML might eventually enable evolutionary prototyping, through the systematic replacement of pure functional with imperative constructs. Furthermore, there is good tool support for proving properties of SML programs. Stepney [Ste93] discusses the use of Prolog for implementing

a compiler from DS specified in Z. Diller [Dil90] briefly discusses the alternatives of Miranda and Prolog for animating Z.

In the late 1980's, Murray [Mur89] explored the construction of a SML tool set for Z, making substantial use of the SML functor mechanism. While this demonstrated the feasibility of prototyping Z in SML, Murray did not elaborate a detailed methodology.

3 From Z to SML

Our translation from Z to SML is based on the simple correspondences between Z and SML types shown in Figure 1.

Z	SML
logical constants	`bool`
$\mathbb{N}$ and $\mathbb{Z}$	`int`
real numbers	`real`
$Free ::= Option1 \mid Option2$	`datatype Free = Option1 | Option2`
arbitrary constant	`string`
$X \times Y$	`X * Y`
$\mathbb{P}\, S$	`S list`
$X \leftrightarrow Y$	`(X * Y) list`

Fig. 1. Z/SML type correspondences.

Rather than requiring the full translation of Z schema to problem specific SML, like Murray we provide SML libraries of polymorphic higher order functions to support the standard Z set, relation and function operations. See Figure 3.

Then, to construct a prototype:

- SML types and type aliases are used to define Z types;
- the state schema is translated to a boolean assertion function;
- for each schema, the pre-condition is translated to a boolean function and the post-condition is translated to a function returning a tuple of all state variables;
- the total schemas are combined in a single function of associated free type values;
- the total scheme function is called by a wrapper function that tests the state schema assertion as an invariant and raises exceptions appropriately;
- test cases are devised to satisfy and fail all pre-conditions, and the invariant assertion.

For example, consider the schema shown in Figure 3 for withdrawing a quantity of an item from a warehouse[Lig01].

Z	SML
$x \in S$	`member x S`
	`member : ''a -> ''a list -> bool`
$\forall x : S \bullet p(x)$	`all p S`
	`all : ('a -> bool) -> 'a list -> bool`
$\exists x : S \bullet p(x)$	`exists p S`
	`exists : ('a -> bool) -> 'a list -> bool`
$S1 \subseteq S2$	`subset S1 S2`
	`subset : ''a list -> ''a list -> bool`
$S1 = S2$	`equals S1 S2`
	`equals : ''a list -> ''a list -> bool`
$S1 \cup S2$	`union S1 S2`
	`union : ''a list -> ''a list -> ''a list`
$S1 \cap S2$	`intersect S1 S2`
	`intersect : ''a list -> ''a list -> ''a list`
$S1 \setminus S2$	`diff S1 S2`
	`diff : ''a list -> ''a list -> ''a list`
$a \mapsto b$	`(a,b) : A * B`
$A \leftrightarrow B$	`(A * B) list`
$\mathrm{dom}\, S$	`dom S`
	`dom : ('a * 'b) list -> 'a list`
$\mathrm{ran}\, S$	`ran S`
	`ran : ('a * 'b) list -> 'b list`
$R(\!\vert S \vert\!)$	`image R S`
	`image : (''a * ''b) list -> ''a list -> ''b list`
R^{-1}	`inverse R`
	`inverse : ('a * 'b) list -> ('b * 'a) list`
$S \lhd R$	`domrest R S`
	`domrest : (''a * ''b) list -> ''a list -> (''a * ''b) list`
$S \lhd\!\!\!\cdot\, R$	`domantirest R S`
	`domantirest : (''a * ''b) list -> ''a list -> (''a * ''b) list`
$R \rhd T$	`ranrest R T`
	`ranrest : (''a * ''b) list -> ''b list -> (''a * ''b) list`
$S \rhd\!\!\!\cdot\, R$	`ranantirest R S`
	`ranantirest : (''a * ''b) list -> ''b list -> (''a * ''b) list`
$f\ x$	`fapply f x`
	`fapply : (''a * 'b) list -> ''a -> 'b`
$f \oplus g$	`over f g`
	`over : (''a * 'b) list -> (''a * 'b) list -> (''a * 'b) list`

Fig. 2. SML library functions for Z operators.

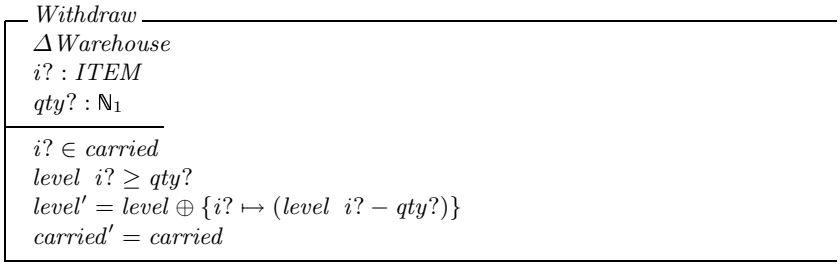

Fig. 3. Z schema for withdrawing quantity of item from warehouse

Here, $level : ITEM \mapsto \mathbb{N}_1$ records how many of each item are held in stock and $carried : \mathbb{P}\, ITEM$ records which items are held.

Types for *ITEM*, *level* and set of *ITEM* are defined as shown in Figure 3. Note that $\mathbb{N}_1$ is represented as `int`.

```
type ITEM = string;
type LEVEL = (ITEM * int) list;
type CARRIED = ITEM list;
```

Fig. 4. SML representations for Z warehouse types

The schema may be translated as shown in Figure 5.

```
- fun preWithdraw (carried:CARRIED,i:ITEM) =
       member i carried andalso
       fapply level i >= qty;
> val preWithdraw = fn : CARRIED * ITEM -> bool

- fun doWithdraw (level:LEVEL,carried:CARRIED,i:ITEM,qty:int) =
       (over level [(i,fapply level i - qty)],
        carried)
> val doWithdraw = fn : LEVEL * CARRIED * ITEM * int -> LEVEL * CARRIED
```

Fig. 5. SML for withdrawing quantity of item from warehouse.

4 Discussion

The new module was taught for the first time in 2002, to 56 students. Module assessment included a practical exercise based on a simple air traffic control example [Lig01], specified as 3 total schema built from 8 auxiliary schema. The example is actually incompletely specified: most of the students spotted and corrected this while constructing their prototypes. Almost all of the students successfully completed the exercise. At the end of the module, implementation in Java from an SML prototype was briefly discussed.

Students were asked to comment on how this approach compared with their previous preference, which most nominated as procedural implementation in Java. Overall, the students were positive about prototyping Z in SML, often commenting that it seemed to reduce errors and development time, and increased assurance that the software met the specification. Many also commented that prototyping had greatly increased their command of Z. Some suggested that the approach might be better suited to much larger problems. However, a minority remained unhappy, preferring the direct Java hack.

We are pleased that this approach appears to have made formal specification and declarative programming more accessible to our students. However, the approach, while formally motivated is clearly informal. It might be placed on more solid foundations by proving the SML libraries correct relative to standard set theoretic definitions of the Z operators, and then investigating the use of a theorem prover to try and establish the correctness of SML prototypes relative to Z specifications. It would also be interesting to look at building tools to assist and automate the approach.

Acknowledgments

I would like to thank Ken Robinson, Julian Richardson and Hunter Davis for fruitful discussion. I would also like to thank my students for constructive feedback on the 2nd year *Prototyping from Specification* module.

Further details may be found in:
`http://www.macs.hw.ac.uk/~greg/F22HW3`.

References

[Abr96] J-R. Abrial. *The B Book*. CUP, 1996.

[Dil90] A. Diller. *Z: an Introduction to Formal Methods*. Wiley, 1990.

[HJ89] I. J. Hayes and C. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[Kal90] A. Kaldewaij. *Programming: the derivation of algorithms*. Prentice-Hall, 1990.

[Lig01] D. Lightfoot. *Formal Specification Using Z*. Palgrave, 2001.

[Mic93] G. J. Michaelson. *Interpreter Prototypes from Formal Language Definitions*. PhD thesis, Heriot-Watt University, 1993.

[Mor90] C. Morgan. *Programming from specifications.* Prentice-Hall, 1990.

[MTH97] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Mur89] J. Murray. ML Prototypes from Z Specifications. Master's thesis, Heriot-Watt University, 1989.

[Sco72] D. Scott. Lattice theory, data types and semantics. In R. Rustin, editor, *Courant Computer Science Symposium 2: Formal semantics of programming languages*, pages 65–106, 1972.

[Spi89] J. M. Spivey. *The Z Notation.* Prentice Hall, 1989.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRG-19, Computer Laboratory Programming Reserach Group, University of Oxford, August 1971.

[Ste93] S. Stepney. *High Integrity Compilation.* Prentice-Hall, 1993.

[Sto77] J. E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory.* MIT Press, 1977.