Declarative Reasoning about the Structure of Object-Oriented Systems

Roel Wuyts
Programming Technology Lab
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
E-Mail : rwuyts@vub.ac.be
WWW: http://progwww.vub.ac.be/∼rwuyts

Published in the Proceedings of TOOLS-USA'98 [1]

Abstract

The structure of object-oriented systems typically forms a complicated, tangled web of interdependent classes. Understanding this implicit and hidden structure poses severe problems to developers and maintainers who want to use, extend or adapt those systems. This paper advocates the use of a logic meta-language to express and extract structural relationships in class-based object-oriented systems. As validation the logic meta-language SOUL was implemented and used to construct a declarative framework that allows reasoning about the structure of Smalltalk programs. The declarative framework's usefulness is illustrated by expressing different high-level structural relationships such as those described by design patterns.

1: Introduction

When developing object-oriented systems many design techniques are used to ensure that the result is reusable, extensible or maintainable. These techniques are implemented by distributing responsibilities over several objects. As a result, the structure of the implemented system is a complex web of communicating classes [8], where each class implements one or more roles from the design. Reusing or maintaining the system is very complicated because the link between the design and the implementation is missing.

While many documentation techniques for object-oriented systems exist, very few document the structure of the system or the link between design and implementation. Design schemes and reference manuals provide only global overviews of this structure, mostly without referencing the actual implementation. On the other hand, an API or an interface description provides such deeper insight, but without the global picture.

More recent techniques like design patterns provide combinations of the two, giving design information while not completely ignoring implementation aspects. However, they

[1] Copyright 1998 IEEE. Published in the Proceedings of TOOLS-26'98, August 3-7, 1998 in Santa Barbara, California. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

are only available on paper in a reference guide. Cookbooks or tutorials provide only a very limited view and are more targeted towards learning the basic features of the system.

Because the link between the global information and the local implementation is missing, the impact of a local change on the global design and vice versa is not clear. While the first gives rise to propagation of change errors, the second makes it difficult to assess the impact of necessary changes. The typical work-around for both is to manually extract the structure from the code. In the case of a local change, a bottom-up approach is followed by tracing the impact of the changes through the framework, a tedious and error-prone process. Global design changes require a top-down analysis starting from the design to make the changes. Because of the size and complexity of current software, this is hard to do, and overlooking or misunderstanding one concern that was handled in the old implementation might compromise the entire structure. A solution to both problems could be to express and extract the structure in a high level medium.

Another example of information that is currently not taken into account is programming conventions that are followed throughout the system [2]. Such conventions are usually adopted when developing a large system, for example 'every persistent class should implement a method *store* to serialise it'. While such conventions can be written down in a manual, it is difficult - except for the trivial ones - to enforce them or find the places in the system where they are breached. These violations might indicate errors, in which case they have to be dealt with. They might also be deliberate decisions, in which case they are very important design documentation. A solution to this problem is expressing the necessary conventions in a medium that allows infringements in the implementation to be found.

The common denominator in the sketched problems is the incapability to express high-level structural information in a computable medium that is then used to extract implementation elements. To solve this problem we introduce a logic programming language as meta-language to express and reason about the structural information of software systems.

Writing down the structure in a meta-language has two main advantages. First, the information is available in executable form in the programming language. This means that tools can be built which use this information. Second, there is no need to include the structural documentation in the code. It is sufficient to specify the structural relationships in the meta-language to extract elements from the code. For example, by expressing a design pattern structure it is possible to extract the participants of the pattern instead of having to manually document every participating class. It's even more advantageous to use a logic programming language as meta-language because this is an open and powerful medium that permits reasoning on a high-level of abstraction.

We have validated our approach by implementing the logic meta-language SOUL (Smalltalk Open Unification Language) in VisualWorks Smalltalk. We proceeded by expressing basic relationships commonly encountered in class-based object-oriented systems, building a declarative framework of facts and rules. This was then used to write down higher-level relationships like those expressed by design patterns, and it was used to experiment with the definitions on code from the Smalltalk libraries.

The rest of this paper is structured as follows: first we will introduce logic programming and SOUL. Than we will describe the declarative framework that covers the basic properties of class-based object-oriented languages. We will then introduce the higher-level relationships which are applications of the basic rules, and the future work that looks at some other applications. The last topic before the conclusion discusses related work.

2: Logic Meta-Programming and SOUL

Non-declarative programs consist of data structures and control structures. Declarative programming languages differ from this scheme by only needing data declarations and using one implicit and general control structure. Logic programming languages are declarative languages where the data is represented by Horn clauses and where logic inference is the only control structure. Facts are used to specify static information which is always true in the application domain. Rules are used to derive new facts from existing ones. The premise of a rule specifies the conditions under which a new fact can be concluded. Queries are used to interrogate this information base.

Logic programs have several attractive properties. First, the absence of control structures result in a simple, easy to learn and easy to use language. Another appealing feature is the multi-directionality, which means that real mathematical relations are described that have no notion of input or output parameters and that can be interpreted in many ways. A third concern is the power of logic programming. Unlike other query languages like SQL or regular expressions, logic programming is Turing equivalent. A final, very important property is that logic programming languages are very open, which means that everybody can easily add information to use in the unification process. The major shortcoming of logic programming languages is the sometimes slow execution time, depending on the query that needs to be solved.

Since this paper advocates the use of a logic programming language as meta-language we have somehow to represent base-language programs in the logic programming language. The term meta-language is used for a language that allows the reasoning about another language, the base language [13]. We will do this by using a general parse tree representation, which enables the use of fine-grained static information. All facts and rules use these parse trees, making them undependable of a particular base-language.

As validation we implemented a logic programming language in VisualWorks Smalltalk. This language, SOUL (Smalltalk Open Unification Language), is based on PROLOG (the most widely known logic programming language) [12], but has an extension that allows unification on user-defined elements expressed in Smalltalk. Because parse trees are integrated, by using tagged lists, SOUL is a meta-language capable of reasoning about base-language programs.

3: Overview of the Declarative Framework

The goal of this paper is to demonstrate that the structure of an object-oriented system can be described at the meta-level in a logic programming language. This information can then be used to formulate high-level facts and rules that allow reasoning about code. For this we constructed a declarative framework consisting of two layers of rules.

It is important to stress that the facts and rules from the first layer express basic information about the structure. Therefore they act as the core of the framework adaptable to particular environments. The next sections will present these two layers in more detail.

3.1: Basic Layer

class(?class)	the classes
superclass(?super,?sub)	the inheritance relationship
method (?class,?m)	the methods of a class
methodSelector(?m,?sel)	the name of a method
selector(?c,?sel)	the names of methods in a class
parsetree (?m,?tree)	the parsetree of a method
instVar(?class,?var)	the instance variables of a class
abstractMethod (?m)	defines abstract methods

Table 1. Representation Rules

Representation Rules. The representation rules (see table 1) deal with the representation of elements of the object-oriented language (such as classes, methods and statements) in the logic meta-language. They all make frequent use of Smalltalk expressions to get to the base system, thus bridging the two worlds. Therefore these rules are essentially base-language dependent, while the rest of the layers based on the representation rules are language independent.

Before giving an example, first a note about basic SOUL syntax: logic variables are denoted with question marks, the comma is used for the boolean *and*, and terms between square brackets are *Smalltalk terms*, terms that contain special Smalltalk expressions which can refer to logic variables.

As a first example, we introduce the *class* predicate, consisting of two rules:

```
Rule
    head: class(?C)
    body: constant(?C), [Smalltalk includes: ?C name].

Rule
    head: class(?C)
    body:
        variable(?C),
        generate(?C,[Smalltalk allClasses]).
```

The first rule describes what happens when the head is invoked and ?C has a value assigned: the Smalltalk term serves as a predicate that returns whether or not ?C's value is a class (note that Smalltalk is a global variable in the VisualWorks environment that, among others, holds the classes). The second rule generates classes by using the SOUL system predicate *generate*, using the second argument (another Smalltalk term) to enumerate all classes and to bind each of the results to the first argument, ?C.

With these two rules defined, we can ask the meta-system to check whether Collection is a class (it is put between $\sim$ because it is a constant containing a Smalltalk expression, in this case an expression that returns the class itself):

```
Query class(~Collection~)
```

in which case it will return true, or we can ask the meta-system to list all classes:

```
Query class(?X)
```

after which there will be about a thousand solutions for ?X, namely every class in our system.

abstractClass (?c)	class ?c is an abstract class
commonMethods(?c1,?c2,?m1,?m2)	method ?m1 in class ?c1 has the same name as method ?m2 in class ?c2
commonSelectors(?c1,?c2,?sel)	both class ?c1 and class ?c2 have a method named ?sel
implements(?c,?sel)	class ?c implements a method named ?sel
rootClass(?c)	class ?c is a root class
arguments(?m,?args)	the list of arguments ?args of method ?m
statements(?m,?stats)	the list of statements ?stats of method ?m
temporaries(?m,?temps)	the list of temporaries ?temps of method ?m
hierarchy(?root,?sub)	all subclasses ?sub of class ?root (direct and indirect)
isReceiver(?r,?stat)	all receivers ?r in a statement ?stat
isSendTo(?r,?sel,?stat)	statement ?stat contains sends of a message ?sel to a receiver ?r
subclass(?sub,?super)	?sub is a subclass of ?super
understands(?c,?sel)	class ?c understands a message with name ?sel

Table 2. Basic Structural Rules

By using Smalltalk terms it is unnecessary to explicitly specify for each class in the object-oriented system that it is a class in our logic meta-system. Instead, we only define the relationship (the class rules) in our meta-language and then use this relation to extract the actual elements (the Smalltalk classes) from the system. The other basic predicates from the table are defined in the same manner.

We can already do some more advanced queries. For example finding common superclasses for two classes using the *hierarchy* rule:

```
Query
    hierarchy(?theSuper,~WeakDictionary~),
    hierarchy(?theSuper,~SortedCollection~)
```

This query first finds all the superclasses for a class WeakDictionary, and keeps the results in the variable ?theSuper. It then checks each of these superclasses to see whether it is a superclass of the class SortedCollection. Only the common superclasses remain when this query finishes. When it is often used we could make this query into a rule as follows:

```
Query
    Rule
        head: commonSuperclass(?class1, ?class2, ?common)
        body:
            hierarchy(?common, ?class1),
            hierarchy(?common, ?class2).
```

Basic Structural Rules. The basic structural rules (see table 2) are defined on top of the core rules, and provide extra functionality to deal with the basic structure of class-based object-oriented systems which are mainly inheritance, acquaintance relations and

some common annotations. Rules defined in this set are used throughout the rest of the framework and allow one to perform basic queries about the system.

We can now extract more difficult structures, for example all methods that override abstract methods and do a super send in their body:

```
Query
    hierarchy(?root,?subclass),
    commonMethod(?root,?subclass,?Method,?overriddenMethod),
    abstractMethod(?Method),
    statements(?overriddenMethod, ?stats),
    findAll(
        isReceiver(variable(~'super'~), ?stat), ?stat, ?stats)
```

4: Applications of the Basic Layer

This section describes the different applications that were made using the rules from the basic layer. All the applications described in next sections are base-language independent, because they only fall back on the representation rules. First we describe how rules can be used to help finding structural information. The other applications define rules that express higher-level structural relationships and form the second layer of the declarative framework. First rules are discussed which define how certain design decisions can be implemented in code. Then we have the programming style rules that express the developer's way of working. Last come the rules to formulate structures described by design patterns.

4.1: Advanced Structure Searching

As we stated in the introduction, object-oriented systems are very complicated and tangled. We mentioned that users new to the system need to dive into the code in order to extract the structure, with only some basic examples and an API to begin their quest. They have to find their way through the code, manually tracing through the system and building up the structure as they discover the system.

Using a logic meta-language can assist in this discovering in different ways because the search process is done at a higher level. Firstly, the initial developers might already have expressed parts of their structure, which means that a new user can be assisted by some advanced queries for exploring the code instead of going manually through it all by himself. Secondly rules that are discovered during the search but are not yet described can be added, thereby extending the initial documentation. A new user might for example search for all the abstract classes in the system:

```
Query abstractClass(?C)
```

or alternatively for all the abstract methods and the classes they belong to:

```
Query abstractMethod(?C,?M)
```

At some point in the search, the user might zoom in on one particular class and get all the messages that are sent to an instance variable, using this to find all classes that understand all these messages:

```
Query
```

```
instVar(?class, ?var),
isSendTo(variable(?var), ?method, ?message),
understandsAll(?varClass, ?message).
```

Or he might find double dispatch schemes by looking at all methods in a class, retrieving all the messages and the receivers they are sent to, getting the classes they can be sent to, and ensuring that the called method actually sends back the double dispatch message:

```
Query
    class(?c),
    methods(?c,?m),
    selector(?m, ?mName),
    isSendTo(?receiver, ?sendMessage,?m),
    receiverClass(?receiver, ?class),
    selector(?class, ?sendMessage, ?method),
    isSendTo(?otherReceiver, ?mName,?method),
    receiverClass(?otherReceiver,?c).
```

4.2: Implementation Strategies Rules

Implementation strategy rules describe how certain design-level structures, like one-to-many relationships, can be implemented. It should be stressed that the rules do not describe every possible way a certain structure can be expressed, but only certain specified cases. For example, in Smalltalk we could define that there is a one-to-many relation if some method enumerates a collection held in an instance variable. Because of the openness of the logic meta-language, it is easy to describe other representations. The one-to-many rule is given here as an example, and will be used later on in the composite design pattern:

```
Rule
    head: oneToManyStatement(?method, ?instVar)
    body:
        statements(?method,?stats),
        hasEnumerationStatement(?stats,?enumStatement),
        receiver(?instVar, ?enumStatement).
```

Note that the above rule uses a rule *hasEnumerationStatement* that relies on facts which define messages that are used for enumerations. For our current experiments we used *do:*, *collect:*, and *select:*, but this could easily be changed by adding or removing facts:

```
Fact enumerationStatement(~#do:~).
Fact enumerationStatement(~#collect:~).
Fact enumerationStatement(~#select:~).
```

In a similar manner we can express other multiplicity relationships.

4.3: Programming Style Rules

The programming style rules define the programming style that is used throughout the object-oriented system. Clearly, users want to add rules describing their "way of programming" and express conventions they have used throughout the system. These conventions are typical examples of information which is very important in understanding the system,

but is currently implicit for most systems. For example, in event-driven real-time systems it may be that *postEvent* and the message loop are consistently used (instead of regular message passing).

A different convention could be: *never access instance variables directly, but always through accessor methods.* The exact form of what an accessor method looks like could then be defined. In the simplest case, an accessor method takes the form of a method that simply returns the value of the instance variable. However, one could also use lazy initialisation, or an embedded SQL-query to fetch the value from a database. The simplest form will be presented here as example:

```
Rule
    head: accessor(?method)
    body:
        selector(?method,?name),
        statements(?method,list(return(variable(?name)))).
```

This rule specifies the Smalltalk convention that the method body of an accessor should consist of one statement, namely a return statement that returns an instance variable with the same name as the selector. C++ users could for example change this to use messages prefixed with *get*.

We can then check to see where the system violates this rule, and thus where instance variables are used directly instead of by their accessors:

```
Query
    class(?C),
    instanceVariable(?C,?instVar),
    method(?C,?M),
    not(accessor(?M)),
    statements(?M,?statements),
    isSendTo(?instVar, ?statements,?offendingSend)
```

If this query does not return false, ?C will give the class of the offending method, ?M the method of ?C where the offending call is, ?instVar the instance variable that is called directly and ?offendingSend the message and arguments that are sent to ?instVar. Using this information, the violations can be checked. Other breaches worth checking could be for example that every abstract method is overridden, or that every class in a double dispatch scheme implements the correct messages.

4.4: Design Pattern Structure Rules

Another application of the basic rules is expressing structures such as those described by design patterns. As stated in [7], design patterns capture solutions to common problems which are encountered while designing software. They are the result of recording experience in designing object-oriented software in a form that people can use effectively. Design patterns are always expressed in a consistent format. They all have a name, intent, motivation, applicability, structure, participants, collaborations, consequences, implementation and sample code.

In general, a pattern is detectable if its template solution is both distinctive and unambiguous [4]. We use the declarative framework described in previous sections to express

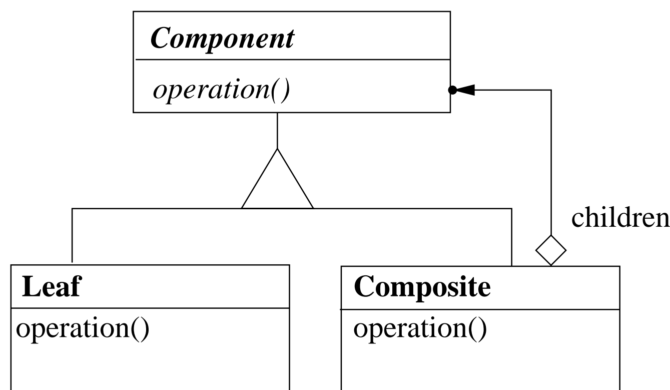

Figure 1. Composite Pattern Structure

structures described by design patterns in an open, formal, non-constraining way. We will give the example of the composite pattern as a blueprint for the other patterns expressed.

As described in [7], the composite pattern is used to compose objects into tree structures to represent part-whole hierarchies. Important for clients who use these structures is that they need not know whether they are using a component or a composite component. As can be seen in figure 1 (based on the picture found in [7]), the structure of a composite pattern is not very difficult. The *composite* class is a subclass of the *component* class that has a one-to-many relation with the component, and where the typical implementation of an overridden method *operation* consists of recursively calling *operation* on the *children*.

We can express this pattern as follows:

```
Rule
    head: compositePattern(?comp,?composite,?msg)
    body:
        compositeStructure(?comp,?composite),
        compositeAggregation(?comp,?composite,?msg).
```

This rule says that a composite pattern consists of a certain structural relationship between the component and the composite, and that there is an aggregation relationship between these two. Each of these rules is given and commented below.

The compositeStructure rule defines that ?comp is a class, and that ?composite is a subclass, direct or indirect, from the composite:

```
Rule
    head: compositeStructure(?comp,?composite)
    body:
        class(?comp),
        hierarchy(?comp,?composite).
```

The aggregation is more complicated to express. It basically boils down to saying that the composite should override at least one method of the component, and in this overridden method it should do an enumeration over the instance variable that holds these composites and recursively apply the method to each of the composites. We can therefore use the oneToManyStatement rule again that was defined in a previous section. The aggregation rule follows:

```
Rule
    head: compositeAggregation(?comp,?composite,?msg)
    body:
        commonMethods(?comp,?composite,?M,?compositeM),
        methodSelector(?compositeM,?msg),
        oneToManyStatement(?compositeM,?instVar,?enumStatement),
        containsSend(?enumStatement,?msg).
```

For example we can use the composite design pattern rule to look for all the possible composite classes as defined above, where the component is VisualPart:

```
Rule
    Query compositePattern(∼VisualPart∼,?comp,?sel)
```

From about 170 subclasses of this class, we found that *VisualComposite* conforms to the described composite structure. As the name indicates, this is indeed a composite class. With the same definition we found composite classes in other hierarchies in the Smalltalk system (like for example *SequenceNode*, a composite class in the *ProgramNode* hierarchy). This validates the description of the structure of the composite pattern as presented here. We do not however pretend to automatically find all composites. *CompositeFont* for example, which is a composite class according to [1], is not found because this class has no overriden operations that recursively call the children.

Because the meta-language is open, users could experiment with other definitions, for example to require that every operation that is overridden in the composite is recursively called on the children, instead of at least one. They could also experiment with other enumeration methods. This demonstrates that the extraction process is an interactive process that helps the developer in finding information, and not completely automatic.

5: Future Work

The goal of this research is to present a declarative framework that can be used to express and extract the structure of object-oriented systems. Therefore we are currently expressing more high-level rules, and are going to support another base language than Smalltalk, thereby ensuring base-language independency. We would also like to experiment with generating code based on the results of queries. Next paragraphs further discuss these topics.

First we want to expand the declarative framework to provide users with a more complete system they can use to reason about code. To accomplish this, we are expressing more relationships, like other design patterns and implementation structures. We will use the *PSI* framework [5] as case study, what will help in iterating over the declarative framework and gaining insight in the rules that are really needed.

Another interesting experiment will be to support another class-based object-oriented language like Java or C++. Because of the design of our declarative framework this will have implications on the representation rules and perhaps on the representation of the parse trees. The representation rules that use Smalltalk terms to retrieve structural elements from the base-language, like the class predicate presented in section 3.1.1, will have to be rewritten to read these elements from the source files. Since we currently use Smalltalk as model for the language, it could also be necessary to adapt our parse tree representation,

to deal for example with multiple inheritance. Probably we will then need to refactor some of the other rules as well, for example the ones that deal with Smalltalk conventions. The result will be a better declarative framework that is really base-language independent.

We also would like to investigate in more detail structural find/replace. Structural find/replace is a common name for operations where parse trees are replaced by other parse trees. Thus new code is generated based on the result of a query. This could be used to support refactoring operations [10], or for building sophisticated code porting tools. This will require serious extensions to the language.

For example, a find/replace statement to replace every direct access to an instance variable with a call to the accessor method (*self* followed by the name of the instance variable and without arguments) could look like this:

```
  find
     systemClasses(?C),
     instanceVariable(?C,?instVar),
     method(?C,?M),
     not(accessor(?M)),
     statements(?M,?statements),
     isSendTo(?instVar, ?statements,?offendingStatement),
     message(?offendingMessage,?offendingStatement),
     arguments(?offendingMessageArgs, ?offendingStatement),
  replace
     ?offendingStatement
by send(
     send(~'self'~,?instVar,list()),
     ?offendingMessage,
     ?offendingMessageArgs)
```

6: Related Work

This section discusses several existing approaches for representing design, architecture or structure in a high-level computable, embedded medium. Next sections describe in more detail some related research efforts, and the differences with our approach.

In [3] Architectural Fragments are introduced. Architectural fragments are descriptions of architectural specifications consisting of a number of roles and initialisation code. The roles can describe classes ranging from only the interface to the full implementation. When the architecture described by the fragments is instantiated, the software engineer manually links roles to domain classes. Every role is thus superimposed with a domain class, resulting in a new class with the same interface as the domain class but conforming to the role it plays in an architecture. Besides the focus on generation of code, this approach introduces a completely new language LayOM, requiring both the architectural fragments and the base-code to be implemented in this new language.

[6] constructed tools for design patterns that are based on fragments and allows developing on different levels of abstraction in the same environment. Three different ways of instantiating and binding fragments to code were identified: top-down (creating a new instance of a pattern), bottom-up (promoting existing code to a pattern) and mixed. However, the research presented in this paper focuses on a fourth aspect, namely extracting

implementation elements that conform to descriptions given in the logic meta-language. This issue is currently missing in the tools offered in [6].

Generic Fuzzy Reasoning Nets (GFRN) [9] is a graphical and executable language developed to model and apply uncertain reverse engineering knowledge. The original contribution is that it explicitly deals with fuzzy knowledge, allowing reverse engineers to add confidence parameters that are taken in account when the legacy code is analysed. While this is necessary for reverse engineering purposes, it also adds considerable overhead in terms of understandibility.

The structure of programs can also be expressed in the language PROGRES [11], a graph rewriting system that is also targeted towards generating code and does not allow extraction of structure.

Our approach differs from other efforts described above because we focus on describing structural elements in the meta-language in order to permit extraction of the structure from the implementation. Most other approaches focus more on generating code from high-level descriptions.

7: Conclusions

Object-oriented systems keep getting more complex in order to address important problems like reusability, adaptability, and so on. Due to the explosion of inherent complicated design and implementation techniques, it becomes more and more complicated to reuse systems. This paradox results from the lack of proper documentation of a system's structure in a computable medium.

This paper addresses the problem using a logic meta-language which allows one to express, to reason about and to extract a system's structure in a base-language independent way. The validation of this statement was done by implementing SOUL, the Smalltalk Open Unification Language, and by developing a declarative framework aimed at reasoning about Smalltalk code. This framework consists of two layers: a first language dependent layer that allows basic reasoning about the system, and a second language independent layer that uses the first layer to provide high-level reasoning.

Finally we validated the declarative framework by demonstrating its application, namely performing advanced structural searches, finding violations of programming conventions and expressing structures used in design patterns and implementation strategies.

8: Acknowledgements

I wish to thank my promoter Theo D'Hondt and Patrick Steyaert for our fruitfull discussions which led to this paper. Further more I would like to thank my colleagues at the Programming Technology Lab for proof-reading and discussing this paper: Koen De Hondt, Wolfgang De Meuter, Kris De Volder, Carine Lucas, Kim Mens, Tom Mens and Tom Tourwé. Last but not least I also thank Stephane Ducasse from the Software Composition Group at the University of Berne for proofreading the paper and getting the LaTeX'ed version right. This research was conducted with a doctoral grant from the Instituut voor Wetenschap en Technologie (Flanders, Belgium).

References

[1] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion.* Addison-Wesley, 1998.

[2] K. Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, Upper Saddle River, 1997.

[3] Jan Bosch. Specifying frameworks and design patterns as architectural fragments. Technical report, University of Karlskrona/Ronneby, 1997.

[4] Kyle Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, 1996. TR-96-07.

[5] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. From custom applications to domain-specific framworks. *Communications of the ACM*, 40(10):71–77, October 1997.

[6] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495, Jyväskylä, Finland, 9–13 June 1997. Springer.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, Reading, Massachusets, 1995.

[8] A. Goldberg and K.S. Rubin. *Succeeding with objects. Decision Frameworks for Project Management.* Addison-Wesley, 1995.

[9] Jens Jahnke, Wilhelm Schaefer, and Albert Zuendorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 193–210. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.

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

[11] A. Schürr, A. J. Winter, and A. Zuendorf. Graph grammar engineering with PROGRES. In W. Schäfer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, pages 219–234. Lecture Notes in Computer Science Nr. 989, Springer-Verlag, September 1995.

[12] L. Sterling and E. Shapiro. *The art of Prolog.* The MIT Press, Cambridge, 1988.

[13] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussel, 1994.