Vrije Universiteit Brussel
Laboratorium voor programmeerkunde
Faculteit Wetenschappen - Vakgroep Informatica
(Oktober 2000)

Automatische architecturale conformiteitscontrole door middel van logisch meta-programmeren.

Kim Mens

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

Promotor: Prof. Dr. Theo D'Hondt

Samenvatting

In deze doctoraatsverhandeling stellen we voor om de techniek van het 'logisch meta-programmeren' te gebruiken voor de ontwikkeling van een expressieve architecturale taal, met bijhorend algoritme, voor de automatische 'conformiteitscontrole' van de implementatie van een softwaresysteem t.o.v. één of meerdere 'architecturale gezichtspunten'. We bereiken een maximum aan expressiviteit door de architecturale elementen en hun relatie met de implementatie te beschrijven in een logische programmeertaal die kan redeneren over entiteiten en relaties in de implementatietaal. De thesis beperkt zich tot 'statische' conformiteitscontrole. M.a.w., we redeneren enkel over de statische structuur van een software-implementatie, en we beschouwen geen dynamische informatie. Een tweede beperking die we maken is dat we enkel objectgerichte implementaties beschouwen, en Smalltalk-implementaties in het bijzonder.

We verdedigen de thesis in een aantal stappen. Vooreerst presenteren we een elegante en eenvoudige, doch expressieve, architecturale taal, met daar bovenop een algoritme om automatisch architecturale conformiteit te controleren. De nadruk wordt hierbij gelegd op de onafhankelijkheid van de architecturaal formalisme t.o.v. een beschouwde architectuur en implementatie. Ten dele is de aanpak zelfs onafhankelijk van de gekozen implementatietaal. Om de haalbaarheid van het algoritme na te gaan, implementeren we een prototype van een ondersteuningsprogramma voor conformiteitscontrole. Gebruik makend van dit prototype wordt een gevalsanalyse uitgevoerd op een bestaande middelgrote Smalltalk-applicatie, die zo'n 100 klassen bevat. Steunend op de resultaten van deze gevalsanalyse, worden enkele toekomstige verbeteringen en optimalizaties van het formalisme voorgesteld. Een interessante uitbreiding is bijvoorbeeld een incrementele variant van conformiteitscontrole. Tenslotte, aangezien het ontwikkelde prototype ondersteuningsprogramma nog vrij experimenteel is, bespreken we hoe dit zouden kunnen uitgebreid worden tot een realistisch en in de praktijk bruikbaar ondersteuningsprogramma dat automatische ondersteuning biedt voor architecturale conformiteitscontrole. Nog verder extrapolerend argumenteren we enkele van de gewenste karakteristieken en eigenschappen voor een ideale volgende-generatie architectuurgestuurde software-ontwikkelingsomgeving.

De voornaamste bijdragen van deze doctoraatsverhandeling zijn, in volgorde van belangrijkheid:

1. We stellen een algemene en expressieve architecturale taal voor, met bijhorend algoritme, om automatisch conformiteit van de implementatie van een software systeem t.o.v. zijn architecturale gezichtspunten te kunnen controleren;

2. We tonen aan dat de expressieve kracht van de techniek van het logisch meta-programmeren een software-architect toelaat om de afbeelding van architecturale elementen op implementatieconstructies uit te drukken op een zeer expressieve, doch summiere en intuïtieve manier;

3. We tonen aan dat 'virtuele softwareclassificaties' een krachtige, elegante en intuitieve manier vormen om architecturale abstracties van implementatieconcepten te vatten, en dat 'virtuele afhankelijkheden' een hoog niveau en intuïtief mechanisme zijn om complexe relaties tussen implementateconcepten te abstraheren;

4. We illustreren het nut van het uitdrukken van verschillende overlappende architecturale gezichtspunten die de implementatiestructuur kunnen 'doorsnijden';

5. We tonen aan dat een logische meta-programmeertaal een geschikt implementatiemedium is om het voorgestelde conformiteitscontrole-algoritme en de architecturale taal in uit te drukken;

6. We schetsen hoe het conformiteitscontrole-algoritme zou kunnen verfijnd worden tot een meer incrementele versie.

Vrije Universiteit Brussel
Programming Technology Laboratory
Faculty of Sciences - Department of Computer Science
(October 2000)

Automating Architectural Conformance Checking by means of Logic Meta Programming

Kim Mens

A dissertation submitted in partial fulfillment of the requirements of the degree of Doctor in Sciences

Advisor: Prof. Dr. Theo D'Hondt

When architects of several minds
Sketch their systems with boxes and lines
Their frameworks of objects
Allow all their projects
To share in each others' designs

Mary Shaw [73]

Contents

List of Figures

List of Tables

Acknowledgements

An important part of the research reported on in this dissertation was funded by the Brussel's Capital Region and Getronics Belgium, in the context of a research project on "Compliance Checking in Object-Oriented Systems". The research was conducted at the Programming Technology Lab of the Vrije Univeristeit Brussel.

I am grateful to many persons for their aid and support during the writing of this research dissertation. Without doubt, I am indebted most thanks to Roel Wuyts, for various reasons. Probably, this dissertation would not have seen the light, if it were not for him. Most ideas behind this dissertation stem directly or indirectly from a joint paper we presented at the TOOLS Europe 1999 conference [52]. This boosted both his and my research and led to this dissertation (and will hopefully lead to his as well). He is also the main developer of the SOUL logic language and system, which was used as a case study in this dissertation. In addition, his system was used for some of the experiments I conducted, although I later switched to another logic language. But even then we were able to share and co-develop some logic predicates for reasoning about Smalltalk source code. Finally, I have to thank him for proof-reading some parts of my dissertation; in particular those that were related to SOUL.

I owe much gratitude to all my proofreaders. In particular, I thank my major proofreader and brother Tom Mens for his meticulous comments, and my colleague and friend Kris De Volder for his critical reviews. No matter how well I tried to hide them, they always managed to find the weak spots in my argumentation. Furthermore, Kris' Ph.D. dissertation was my big example. I hope my dissertation will be at least as motivating to other people as Kris' dissertation was to me. I am equally thankful to all other proofreaders for taking the time and pains to work their way through substantial parts of the text: Bart Wouters (with a special thanks for his just-in-time proofreading), Johan Fabry, Tom Tourwé, Johan Brichau and Derek Rayside. I greatly appreciate that Gail Murphy could make the time to read and comment on some chapters I sent her, even though she had a very tight time schedule. And of course a big thanks to all those who supported and helped me when I had to revise the text after it had been reviewed by the Ph.D. committee.

I am especially indebted to my advisor Theo D'Hondt, for his unconditional support and guidance, for providing a very flexible working environment, and of course for reading and commenting on final versions of the dissertation. I also thank all other members of my Ph.D. committee for their extensive reviews and comments, and for making the time to be part of the committee in the first place: Amnon Eden, Viviane Jonckers, Dirk Vermeir, Patrick Steyaert and Miguel Wermelinger.

I appreciate the help of Dirk Bontridder, Koen De Hondt, Alain Grijseels, Carine Lucas, Natalia Romero and Patrick Steyaert for the many fruitful discussions we had on many topics and for their encouragement and support.

I should not forget to thank all my colleagues and ex-colleagues at the Programming Technology Lab for providing a great and motivating working environment. It has been a great pleasure to work with them over the years. Most of them were already mentioned above, others are: Niels Boyen, Wim Codenie, Linda Dasseville, Jan De Laet, Wolfgang De Meuter (with a special thanks for helping me out with part of the formalism), Serge Demeyer, Dirk Deridder, Karel Driessen, Wim Lybaert, Isabel Michiels, Lucas Stoops, Michel Tilman, Werner Van Belle, Marc Van limberghen,

Karsten Verelst and Mark Willems. And of course, I appreciate our secretaries Lydie Seghers and Brigitte Beyens very much for their support.

Last but not least, I want to thank my wife Kathy Van Lindt for putting up with me during this last year and for having provided me with a firm deadline. Unfortunately, I did not even make that deadline (but the baby was still welcome). I thank my new-born son, Nick, for making me see things in the right perspective. And although it may have become one of the most frequently used clichés in dissertation acknowledgements, I thank my parents for having provided me the opportunity to study as well as for their moral support.

Finally, I thank all my other friends, colleagues, ex-colleagues, students, family members, and so on, who explicitly or implicitly supported or helped me (and who never ceased to ask when my dissertation would finally be finished) but were somehow forgotten in the above list.

Chapter 1

Introduction

In this dissertation, we propose an expressive architectural language in which to describe software architectures and their mapping to some software implementation. The language supports the declaration of multiple architectures, called architectural views, on the same software system. Each such views focuses on a different aspect of the structure of that system. In addition to this architectural language, we propose an algorithm which reasons about the descriptions in the architectural language to automatically check conformance of the implementation of some software system to its architectural views. The approach we adopt is a logic meta-programming approach. I.e., we express both the architectural language and the conformance checking algorithm in a logic programming language which is used as a meta-programming language to reason about implementation artifacts in some (object-oriented) base language.

1.1 Thesis

Software architecture is increasingly recognized as an important level of design for software systems. Software architectures describe complex software systems at a sufficiently high level of abstraction that their conceptual integrity and other key system properties can be clearly understood early in the design cycle [47]. An architecture may include multiple views of the same system, each emphasizing a different aspect of that system [11, 65]. In this dissertation, we focus on the problem of checking conformance of an implementation of a software system with its architectural views. More precisely, we make the following claim.

Thesis. *Automated support for checking conformance of an implementation of a software system to its architectural views can be achieved in a very expressive way by adopting a logic meta-programming approach.*

In contemporary software development, and in object-oriented software development in particular, descending from higher levels of abstraction to lower levels (for example, from design to implementation) is relatively straightforward and well supported by software engineering methods and tools. Transition in the opposite direction, however, is not so straightforward and thus less supported. As a consequence, contemporary software development methods often follows a top-down approach, starting at high levels of abstractions that are gradually refined to lower level ones. Once the software starts to evolve, however, in the face of time constraints, modifications are often applied directly to the implementation level, thus sacrificing conformance to and consistency with the information in the earlier life-cycle phases.

Architectural descriptions, which by their very nature seem to cross-cut the implementation, are particularly difficult to enforce. This problem of keeping an implementation in conformance with the architectural descriptions is frequently discussed in research literature using terms such as architectural drift and architectural erosion [35, 65]. Because evolution of an implementation

causes it to drift away from the original architecture, the implementation code should be kept consistent with its architecture. Equally so, when the software architecture itself evolves, one should be able to detect those places in the source code where changes need to be made (and which), so that the code still conforms to the evolved architecture. When an implementation is kept in conformance with the intended architecture, it becomes more maintainable, easier to understand, easier to evolve and reuse, and so on.

Architectural conformance checking is the task of verifying whether the implementation *structure* of a software system corresponds to the more abstract high-level structure described by its software architecture. We define the structure of a software system as the organization of its parts and the interactions between those parts. For now, we restrict ourselves to object-oriented implementations, and Smalltalk implementations in particular.[1] In this context, the parts of a software system are its classes, methods, variables, etc; and the interactions are class instantiation, method invocation, variable access, etc. Another restriction we make is that we consider only the *static* structure of a software system (such as what are the classes, methods and instance variables, and how are they structured), and do not take into account run-time information (such as which objects are active during a run of the program, and how they cooperate).

To provide automated support for architectural conformance checking, we advocate an approach based on *logic meta programming*. In this approach, logic expressions are used at a meta level to qualify an implementation with architectural concerns, and to declare architectural relationships among those concerns. Because these expressions are meta-level descriptions on top of an actual implementation, it is easy to verify conformance of this implementation to these descriptions. Thanks to its powerful concepts of logic unification and backtracking and its multi-way querying facilities, the logic paradigm is a suitable medium in which to implement our conformance checking algorithm. However, it is more than merely a convenient implementation medium for the algorithm. It is also very well suited to describe the mapping of architectural concepts and relations to implementation artifacts and their dependencies.[2] The declarative nature and expressive power of the logic language enables an architect to describe the 'architectural mapping' in a very expressive, yet concise and intuitive, way.

1.2 Motivation

> *Suppose I want to understand the "structure" of something. Just what exactly does this mean? It means, of course, that I want to make a simple picture of it, which lets me grasp it as a whole. And it means, too, that as far as possible, I want to paint this simple picture out of as few elements as possible. The fewer elements there are, the richer the relationships between them, and the more of the picture lies in the "structure" of these relationships.* Christopher Alexander [3]

Software architectures describe the overall structure of a software system, abstracting away all implementation details and focusing only on a few concepts of interest and their relationships. Software architectures facilitate the understanding of large and complex software systems, by providing a simple picture that allows software engineers to grasp the global structure of a system as a whole. When we know the architecture of a software system, it becomes much easier to modify or maintain the system.

Unfortunately, the architecture is not always explicitly documented. Furthermore, even when it is documented, because of the problem of architectural drift, an implementation tends to drift away from the documented architecture, making this documentation unreliable. As a consequence,

[1] In Chapter 8, Section 8.4, we broaden the scope again and explain how to generalize the conformance checking approach to object-oriented languages other than Smalltalk, as well as to other (i.e., non object-oriented) languages or even to design languages.

[2] This 'architectural mapping' is not part of the implementation of the conformance checking algorithm. It is explicitly declared by a software architect and varies with the architectural view and the software implementation under consideration.

the documentation will not be used anymore, so that the implementation drifts away even more. Because the documented architecture is not used, it is not updated anymore. It does not take long for the architecture to become completely outdated, so that it looses all its beneficial properties. Therefore, it is crucial to keep the architectural documentation up to date [4].

Because there is no explicit link between an implementation and its architecture, and because architectural concepts may cut across this implementation, the problem of architectural conformance checking is a non-trivial one. Although some research has been conducted on the topic, most approaches are restricted in the kinds of implementation artifacts and implementation dependencies that can be considered (see 3.2). Furthermore, most approaches do not support cross-cutting mappings from architectural concepts to implementation artifacts. In contrast, the approach taken in this dissertation tries to be as general and as expressive as possible, and enables the declaration of architectural views that cut across the implementation. It also allows the definition of multiple, potentially overlapping, architectural views, thus providing support for separation of concerns at the architectural level.

This dissertation investigates the feasibility of an architectural conformance checking approach that is as expressive and as flexible as possible. We want to restrict neither the complexity of the relationships that can be expressed, nor the kinds of implementation artifacts about which we can reason at the architectural level, nor do we want to restrict how the architectural descriptions can be mapped to an implementation. To achieve a maximum of expressiveness, we express the architectural entities and their mapping to an implementation in a full-fledged logic meta language. The same language is used to implement an algorithm for checking conformance of an implementation to its architectural views.

This dissertation is also important from another perspective. Our successful use of a logic meta-programming approach to solve the problem of architectural conformance checking, confirms our belief that the emerging technique of logic meta programming is highly suitable to build state-of-the-art software engineering support tools, and, in particular, tools that support co-evolution of an implementation and the earlier life-cycle phases. This is a research direction that is under active investigation at the Programming Technology Lab of the Vrije Universiteit Brussel. K. De Volder adopted a logic meta-programming approach for generating source code from meta-level declarations including both high-level logic declarations and low-level source-code fragments [14]. R. Wuyts investigated the use of logic meta programming to enforce or check design information in the source code and to search or browse for certain design constructs in the source [86]. This dissertation has in common with these research efforts that a logic meta-programming approach is used to provide support for co-evolution, i.e., keeping earlier-level life-cycle artifacts synchronized with implementation artifacts [19].

1.3 Approach

We give a quick overview of the approach we will follow to support the thesis statement presented in Section 1.1. The statement itself already provides a clue of the followed approach:

- The medium used for implementing the architectural conformance checking algorithm is a logic language which can reason about artifacts in some (object-oriented) base language.
- Taking advantage of the powerful features of this logic meta language, we define an expressive architecture language for declaring architectures and their mapping to an implementation.
- Based on this architecture language, we design an algorithm and prototype implementation for automated conformance checking.

First of all, we define our architecture language for declaring architectural views as well as their mapping to an implementation. The 'architectural mapping' is based on the notions of *virtual classifications* and *virtual dependencies*. Instead of explicitly qualifying the implementation artifacts with architectural concerns, implicit virtual classifications are used. Computing a virtual

classification yields a set of implementation artifacts that address a similar architectural concern. Virtual dependencies are logic predicates that define high-level architectural relationships in terms of more primitive implementation dependencies. Both virtual classifications and virtual dependencies are powerful abstractions that can take advantage of the full expressive power of the logic meta language.

Next, we define the conformance checking algorithm which combines all information declared in the architecture language to generate a logical expression that can verify architectural conformance. Based on this language and the conformance checking algorithm, a prototype tool is implemented in the logic meta language.

Using this prototype, a case study is conducted on an existing medium-sized Smalltalk application. We describe multiple architectural views on this application, as well as their mapping to the Smalltalk implementation, and check conformance of this implementation to these architectural views. In addition to validating the feasibility and expressiveness of our conformance checking formalism, this case study illustrates the need and relevance of having multiple, potentially overlapping, architectural views that may cut across the implementation.

Based on the results of the case study, some improvements and extensions to the architecture language, conformance checking algorithm and tool are proposed. Amongst others, we discuss how the algorithm could be transformed into a more incremental version. When small changes are made to either the implementation or an architectural view, and under the assumption that the implementation was already in conformance with the architectural view before the changes, the incremental conformance checker only re-checks conformance for those parts of the architecture and implementation that were affected by the change. There is no need to re-check conformance entirely.

1.4 Contributions

Summarizing all this, the main contributions of this dissertation are:

1. We provide a general and expressive formalism, and a prototype implementation, for automated conformance checking of an implementation of some (object-oriented) software system to multiple architectural views.

2. We show that logic meta programming is a suitable technique for implementing the proposed architecture language and conformance checking algorithm.

3. We illustrate that virtual classifications and virtual dependencies constitute high-level and intuitive mechanisms for abstracting architectural concepts from implementation artifacts, and architectural relations from implementation dependencies, respectively. Virtual classifications and virtual dependencies are very expressive abstractions that can make use of the full expressive power of a logic meta-programming language.

4. We illustrate the relevance of providing multiple, potentially overlapping, architectural views that may cross-cut the implementation.

1.5 Organization of the dissertation

In the next chapter, we introduce some terminology that will be used throughout the dissertation, and provide background information on the topics of software architecture, logic meta programming, software classification and separation of concerns. This information is analyzed in Chapter 3 to show that the research problem this dissertation addresses is an important problem that has not been solved before. We also elaborate a bit more on the approach followed. Chapter 4 introduces the case that will be used throughout this dissertation: the Smalltalk implementation of the logic language SOUL.

In the subsequent chapters (5 to 7) we describe how we actually solved the research problem. We do this in three steps: we first explain the details of the architectural language and of the conformance checking algorithm (Chapter 5). Then we show how the proposed formalism was implemented in a logic meta-programming medium (Chapter 6). Finally we apply the prototype implementation to the chosen case (Chapter 7). The results of this case study are used to validate the thesis.

Extrapolating from the experience gained with our prototype implementation, Chapter 8 discusses some of the features and properties an industrial-strength tool for conformance checking should possess. Finally, in Chapter 9, we summarize the contributions and conclusions of this dissertation and mention some more future work.

Chapter 2

Preliminaries

This chapter provides background information on some research topics that are relevant to this dissertation. Since the dissertation is about checking conformance of a software implementation to its architectural views, we start with explaining software architecture in general, and architectural conformance checking in particular. Then we discuss logic meta programming, which is the medium in which we will express our conformance checking formalism. The notion of virtual software classifications, a powerful abstraction mechanism in our formalism, is introduced as well. A final section discusses the need for separation of concerns in software in general, and the need for multiple architectural views in particular.

2.1 Software architecture

2.1.1 Introduction

Software architectures emerged as a natural evolution of design abstractions, as software engineers searched for better ways to understand their software and new ways to build larger, more complex software systems [74]. Software architectures present a high-level view of the structure of a software system, enabling engineers to abstract away the irrelevant details and focus on the "big picture" [46]. It is generally agreed upon that software architectures are high-level design abstractions:

- Software design has two stages: architectural design and detailed design. *Architectural design* is the process of defining a selection of software components, their functions and their interfaces to establish a framework for the development of a software system. *Detailed design* is the process of refining and expanding the software architectural design to describe the internals of the components. [20]
- *Architectural design* specifications describe the general structure of a software system, whereas *detailed design* specifications describe the control flow, data representation, and other algorithmic details within the modules [21].
- *Architecture* provides a framework in which to satisfy the requirements and serve as a basis for the *design* [65].
- As the size and complexity of software systems increase, the *design* and specification of overall structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the *software architecture* level of design. [74]

Because software architectures provide a high-level abstract view of the overall structure of a software system, they provide many benefits with respect to software evolution. Some authors even claim that during software evolution, the software architecture becomes the critical aspect of design [25, 28]. To perform a change to a system effectively, a software engineer needs to have some understanding of the system [84]. An engineer may benefit from understanding the structure of the software system — the organization of the source code into components and the interactions and dependencies between those components — since the difficulties encountered when performing a change are dependent to a great extent on the system's structure [58]. An explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large systems [46].

2.1.2 Definitions

In this subsection, we give informal definitions and examples of the concepts of 'software architecture', 'component', 'connector' and 'architectural style'. Most definitions and examples are borrowed from M. Shaw and D. Garlan's book on software architecture [74].

Architectures

Most definitions of *software architecture* [10, 25, 29, 65, 72] characterize a software architecture as a collection of architectural entities, together with a description of the interactions and relationships among those entities and a set of constraints on these entities and their relationships. The architectural entities are typically referred to as the *components* of the architecture, and the relationships among those entities are often called the *connectors*. The software architecture represents the overall organization of a software system and the global control structure.

Some sources [10, 65, 74] state that, in addition to specifying the structure and topology of the system, the architecture should also show the correspondence between the system requirements and elements of the constructed system, thereby providing some rationale for the design decisions. In this dissertation, however, as in [52, 72], we focus on the structural aspects of a software architecture only.

A concrete example of a software architecture, depicted in Figure 2.1, is that of a rule-based interpreter. As we will revisit this architecture later in the dissertation, we briefly explain some of its key components and connectors here. This architecture, as well as the interpreter architecture of which it is a special case, are explained in more detail in [4, 31, 74].

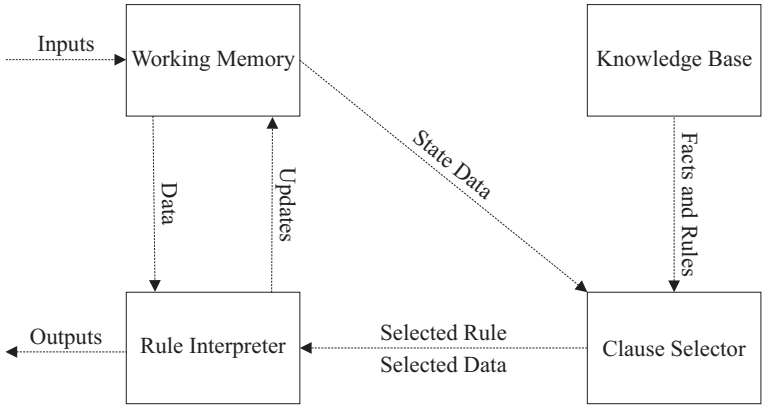

Figure 2.1: The architecture of a rule-based interpreter.

The 'Rule Interpreter' component represents the heart of the inference engine of the rule-based system. It will do the actual interpretation of logic clauses (queries, rules and facts). The 'Working Memory' component represents the current state of the interpretation process (i.e., the

current set of bindings of values to logic variables). The 'Knowledge Base' component represents the logic program that is being interpreted. It contains the rule base and fact memory of the rule-based system. The 'Clause Selector'[1] component models the control and control state of the interpretation engine, such as the current rule or fact being executed and the clauses remaining to be executed. The connectors are shown as arrows in the picture and represent flow of data. The Rule Interpreter selects a clause from the Knowledge Base via the Clause Selector and based on the selected clause potentially updates the current set of variable bindings in the Working Memory.

Components

According to Shaw and Garlan, architectural components are the primary units of computation and state. Many different kinds of components can be distinguished: 'pure computational' components only perform some computation and have no state, 'memory' components represent a shared collection of structured data, 'manager' components contain state and closely related operations, etc.

Each component has an interface specification that defines its properties, which may include its signatures, functionality, guarantees about global invariants, performance characteristics, and so on. Each is of some type or subtype (e.g., process, memory, filter, server). Not every architectural model allows the definition or usage of different component types, though. For example, in module interconnection languages [68], only one type of components is distinguished, namely the modules (e.g., files, packages, libraries) out of which a software system is composed.

Connectors

Shaw and Garlan define architectural connectors as the 'loci' of relations among components. They mediate interactions but are not things to be hooked up (rather, they do the hooking up). Many different kinds of connectors can be distinguished: procedure calls, dataflow connectors (i.e., interaction through streams of data), implicit invocation (e.g., event systems), message passing, instantiation, etc.

Each connector has a protocol specification that defines its properties, which may include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction (e.g., ordering, performance, etc.). Each is of some type or subtype (e.g., remote procedure call, pipeline, broadcast, event). Again, not every architectural model allows the definition or usage of different connector types. For example, in module interconnection languages the only connectors are the implementation dependencies (definition/use or import/export) between the system modules.

Architectural styles

An *architectural style* defines a family of architectures in terms of a pattern of structural organization. Each architectural style has its own vocabulary of component and connector types, as well as a set of constraints on how the components and connectors can be combined.

As a concrete example, consider the 'pipe and filter' architectural style [74]. The only valid component types are 'filters' which can read or write and transform streams of data. The connector types are 'pipes' which transport data streams between filters. Constraints in the 'pipe and filter' architectural style include that filters can only be connected through pipes. The 'pipeline' architectural style is a refinement of the 'pipe and filter' style which additionally requires that every filter has at most one incoming and outgoing pipe. A concrete instance of this architectural style is a traditional compiler, where the filters represent the different stages in the compilation process: lexical analysis, syntax parsing, semantic analysis, optimization and code generation.

[1] In [74], this architectural element is called 'Rule and data-element selection'.

2.1.3 Problems with software architectures

Architectural erosion and architectural drift

Due to a lack of formalization and tool support for software architectures, there are still many technical problems related to the use of software architectures. One severe and commonly accepted problem is that of architectural erosion and the related problem of architectural drift.

D. Perry and A. Wolf [65] define *architectural erosion* as "violations in the architecture that lead to increased system problems and brittleness". They define *architectural drift* as "a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture". In other words, because the implementation of a software application continually evolves, most software applications tend to drift away from their original architecture, causing the application architecture to erode. When no proper actions are taken to counter this erosion, eventually the software will turn into a legacy system.

C. Jaktman et. al. [35] extend D. Perry and A. Wolf's definition of architectural erosion to include the structure of an architecture. They define the structure of a software architecture as being eroded "when the software within the architecture becomes resistant to change or software changes become risky and time consuming. Erosion can also be exhibited when the software is hard to understand or manage due to an increase in the size and complexity of the code and its structure. Erosion can be a result of poor design decisions made while implementing maintenance changes to the system, or a result of limited architectural understanding during software maintenance which may have constrained the flexibility of the design." They provide a list of characteristics indicating architectural erosion in an evolving software system. Amongst others, the list includes the following indicators:

- The architecture is not documented or its structure is not explicitly known.
- The relationship between the architectural representation and the code is unclear or hard to understand.
- The design principles of the architecture are violated when implementing a particular variant of the software system.

The above characteristics could be resolved by adopting a conformance checking approach, such as the one we will propose, where the architecture is explicitly documented and source code can be automatically checked for conformance to the architecture, based on an explicit declarative mapping from this architecture to the code.

Architectural mismatch

In addition to the above problems of architectural erosion and architectural drift, Shaw and Garlan argue that without explicit and formal architectural descriptions, it is difficult to capture the intended architecture of a software system, due to mismatches with the source code. This problem is often referred to as the problem of *architectural mismatch* [26].

> The models implicit in designers' architectural descriptions (both text and diagrams) do not match the actual realization of these models in code. Architectural models are rich, abstract, spontaneous, and almost wholly informal; however, the implementation languages are rigorous, precisely defined, and limited in expressiveness to the constructs of the underlying programming language. As a result of these mismatches, the code fails to capture designers' intentions for the software explicitly and accurately, and precise design documentation does not persist into maintenance. Even insofar as the code captures parts of the design, it does so in a highly distributed fashion, and it is hard for a reader to get a system-level overview. [74]

Our approach addresses this problem of architectural mismatch, by explicitly describing architectures and their mapping to the source code, and by providing a means of checking conformance of the source code to these architectures.

2.1.4 Architecture description languages

The first thing we need to solve the above problems, is a more formal notation in which software architectures can be described explicitly. *Architecture description languages* (ADLs) provide such a notation. The formality of ADLs renders them suitable for manipulation by software tools. This subsection on architecture description languages is largely based on N. Medvidovic and R. Taylor's excellent classification and comparison of ADLs [45].

Over half of the system maintenance effort goes into deciphering what the software actually does, so the inability to record and retain the designers' higher level intentions about component interactions is a major cost generator. Therefore, there is a need for software engineers to explicitly codify their intentions at a suitably abstract level [74]. ADLs provide a formal basis for describing software architectures by specifying the syntax and semantics for modeling components, connectors, and configurations [62].

Many different ADLs can be distinguished, based on the particular problem they focus on, and based on what can be modeled by the ADL. We will not enumerate all possible ADLs here. What is important for this dissertation is a characterization of the essential properties of an ADL. According to Medvidovic and Taylor, an ADL must explicitly model components, connectors and their configurations (i.e., how the components and connectors are interconnected). The interfaces of the individual components should also be modeled. Interfaces for connectors are a desirable (but not an essential) aspect of an ADL. Finally, to be truly usable and useful, and ADL must provide tool support for architecture-based development and evolution.

In many ADLs (though not all) the interfaces of components are specified as *ports*. Any component may have multiple ports, each of them defining a logically separable point of interaction with its environment. The interfaces of connectors are specified as *roles*. Components are *linked* to connectors by linking ports to roles.

What differentiates ADLs from high-level design notations, module interconnection languages (MILs), programming languages and object-oriented modeling notations and languages, is their focus on architecture at a *conceptual* level (as opposed to implementation level) and their explicit treatment of connectors as first class entities. For example, although we already mentioned MILs in the previous subsection as an example of a software architecture model, a MIL is not really an ADL. Its focus is more on the implementation than on the conceptual architecture as it describes the 'uses' dependencies among modules in an implemented system. Furthermore, they support only one type of connection. The boundary between MILs and ADLs is a bit vague, however. Some ADLs, called the *implementation constraining* ADLs directly relate components and connectors to the implementation. Other ADLs, however, are *implementation independent*. They model components and connectors at a high level of abstraction and do not assume or prescribe a particular relationship between an architectural description and an implementation.

2.1.5 Architectural conformance checking

ADLs provide a formal basis for describing software architectures, which makes them suitable for manipulation by software tools. In particular, we are interested in conformance checking tools, which are necessary to solve (amongst others) the problems discussed in Subsection 2.1.3. R. Schwanke et. al. confirm that checking conformance of source code to an architecture is an important problem:

> Practicing architects at Siemens tell us that their most pressing architectural concern is maintaining consistency between the architecture and the code. Whereas it takes a team up to a year to design an architecture, they must then live with it for up to fifteen years of development and maintenance. They also tell us that, by the time an architecture specification is published, it is already wrong. [72]

Architectural conformance checking is about verifying whether some implementation 'conforms' to an architectural description. This is done by defining an architectural mapping that 'refines' the

architectural components and connectors to implementation artifacts and dependencies, and using this mapping to trace whether the implementation actually conforms to the described architectural configuration.

According to N. Medvidovic and R. Taylor [45] as well as P. Clements [11], architectural conformance checking may very well be the area in which existing ADLs are most lacking. Most ADLs do not provide support for refining an architecture to an implementation of a system, nor for checking consistency between an implementation and its architecture.

Some ADLs do allow system generation directly from architectural specifications, thus obliterating the need for conformance checking. However, these ADLs are typically the 'implementation constraining' ones. In other words, they assume that the relationship between architectural elements and those of the resulting implementation is one-to-one. As we will explain later, this is an unreasonable assumption to make.

Although little support currently exists for checking conformance of an implementation of some software system to its architecture(s), below we mention some related and relevant approaches.

- A MIL [68] can be used to formally describe the global structure of a software system, by specifying the interfaces and interconnections among the modules that make up the system. The interconnections between modules are described in terms of the entities they contain (e.g., variables, constants, procedures, type definitions, ...). These formal descriptions can be processed automatically to verify system integrity.

- N. Minsky [55] developed the formalism of *law-governed architecture* which allows the explicit and formal declaration of certain regularities for a given system. These declarations form the 'law' of the system and can be enforced by the environment in which the system is developed. A regularity is similar to an architectural constraint: it is a "global property of a system; that is, a property that holds true for every part of the system, or for some significant and well-defined subset of its parts". A well-known example of a regularity in software is a layered architecture.

  Although architectures can be regarded as a specific kind of regularities, many other kinds of regularities can be expressed as well. N. Minsky's formalism was not specifically targeted towards solving the problem of architectural conformance checking. As the focus of this dissertation is on the particular formalism and technique that are needed for checking architectural conformance, our work is more or less complementary to Minsky's.

- G. Murphy et. al. [57] developed the notion of *software reflexion models* that show where an engineer's high-level model of the software does and does not agree with a source model (that was extracted from the source code), based on a mapping from the source model to the high-level model.

- R. Schwanke et. al. [72] developed *Gestalt*, a language and toolset for specifying software architectures and for checking consistency between the architecture and the code. Both structural consistency and protocol type compatibility at the interfaces are checked.

- As will be explained in Section 2.2, at the Programming Technology Lab some experiments have been conducted to check conformance of Smalltalk source code to design patterns and programming styles [86] and to architectural descriptions [52]. These experiments eventually lead to the conformance checking formalism that is proposed in this dissertation.

A comparison of the similarities and differences among some of these approaches, and more specifically, of how they relate to our approach, is deferred to Section 3.2.

2.1.6 Evolution of software architectures

Software systems have a natural tendency to evolve. This can be due to many reasons: changing requirements, adopting new technology, software maintenance and bug fixing, increasing the efficiency (or other quality aspects) of the software, using the software beyond its original goals, new

insights in the problem domain, new design insights, etc. When a software implementation evolves, conformance checking techniques can be used to verify whether the evolved implementation still conforms to the software architecture.

In addition to the implementation, the architecture itself may (need to) evolve as well. The constantly changing requirements and concerns dictate that we revise the architecture to cope with this. As requirements evolve, so must the architecture *if* it does not already meet the new requirements. Sometimes new requirements are faced that do not significantly affect the architecture itself, although they do force changes in the implementation. However, sometimes we face new requirements and concerns that force us to revise the description of the architecture [66].

The main reason why architectures should evolve is that when the quality of the architecture degrades software modifications become more difficult. This is because design decisions at the architectural level have far reaching consequences on the resultant code and hence on its maintainability [34]. In order to address the problem of architectural drift and to stop or even reverse the effects of software aging, software architectures need to take evolution into account.

Some might argue that support for architectural evolution is not so important because architects could and should try to anticipate possible future changes in advance, and provide hooks for them in the architecture. However, this is not always done due to time pressure. Furthermore, even when architects have provided hooks for future evolution, these hooks are seldom what is needed when the system needs to change. Very few architects have sufficient foresight to anticipate all possible changes, and take these into account.

When the architecture has evolved, conformance checking techniques can be used to check conformance of the original software implementation to the evolved architecture. In the context of evolution (of either the implementation or the architecture), an incremental approach to conformance checking may be most appropriate. With such an approach, instead of re-checking conformance for the entire implementation and architecture, we only need to compare the parts that have changed. We sketch such an approach in Section 8.1.

2.2 Logic meta programming

2.2.1 Logic meta programming at PROG

In this dissertation, a logic meta-programming approach for architectural conformance checking is proposed. This research fits in with the broader research on logic meta programming (LMP) at the Programming Technology Lab (PROG) of the Vrije Universiteit Brussel. The goal of that research is to investigate how the technique of LMP can facilitate the construction of state-of-the-art software development support tools. In particular, a number of experiments have been conducted that use LMP to qualify implementation-level artifacts with enforceable design concerns, including architectural concerns [52, 81, 86]. It seems intuitively clear that design information, and in particular architectural concerns, are best codified as logic constraints or rules. Such rules and constraints can be used to enforce or check design information or architectural constraints in the source code, to search or browse for occurrences of certain design constructs in the source code, or even as a process for code generation and transformation. LMP is an emerging technique, not quite out of the lab as yet. However, it has already been shown to be very expressive [14, 15, 19, 52, 81, 86].

LMP is an instance of hybrid language symbiosis, merging a logic language at meta-level with a standard object-oriented base language. Base-level programs are expressed as terms, facts or rules in the meta level. Logic programming has long been identified as very suited to meta programming and language processing in general; see [15] for related publications. For historical reasons, in this dissertation we concentrate on a Prolog-derivative for our logic meta language; its expressive power and its capacity to support multi-way queries seem particularly attractive. We are not concerned with performance issues at this stage, but we would, at least initially, like as much expressive power on our side as possible.

To conclude our overview of LMP we present some concrete experiments illustrating how it can be used to build state-of-the-art software development support tools. For a more detailed discussion of these experiments, we refer to [19]. This is by no means a complete coverage of LMP, nor even of the experiments conducted at our lab.

- In his Ph.D. dissertation on "Type-Oriented Logic Meta Programming" [14], K. De Volder proposed to use LMP as a way to extend the expressiveness of current type systems. The approach even proved to be sufficiently general, to handle aspect-oriented programming (and even aspect-oriented *meta* programming) as well [15]. K. De Volder used a code-generation approach where code was described at meta level using a mixture of high-level logic declarations and low-level pieces of Java source code. Based on these declarations, his TyRuBa system then generated one or more Java programs satisfying these high-level descriptions. Using the same TyRuBa system, a Master student conducted an experiment to generate the source code of an application by describing it at design level as a configuration of components [67].

- The SOUL system developed by R. Wuyts follows more or less the opposite approach assuming the existence of some repository of source code, on top of which logic meta declarations are defined to reason about this source code. SOUL is a hybrid logic language, implemented in Smalltalk, and with a tight symbiosis with both the Smalltalk language and development environment. Using SOUL, experiments have been carried out to check, browse for, or enforce programming conventions [54], design patterns and styles [86] and to check conformance of source code to architectural constraints [52].

- Adopting accepted design principles and techniques (such as idioms, programming conventions, design patterns and heuristics, and so on) has many advantages when implementing software. Unfortunately, it often results in some performance penalties. To allow for software systems with a clean design, without compromising efficiency, T. Tourwé suggests doing source to source transformation from well-designed implementations to more efficient ones [81]. Again, a LMP approach is put forward. However, in this case, a combination of a logic

and a functional meta language is used. The logic language declares the role certain implementation artifacts play in specific design constructs. The functional language describes the optimization transformations for each specific design construct. These transformations can rely on the information contained in the logic declarations.

2.2.2 Co-evolution

In the previous subsection we introduced the research direction of LMP that is under active investigation at our lab. Most of this research directly or indirectly addresses the need for more control over the evolution of software. In the past, a significant amount of work at the lab focused on the need to document evolution and build conflict detection tools [40, 78] and to formalize the evolution process [53]. More recently, the focus has shifted towards exploring an approach for steering evolution [19]. We have adopted the term *co-evolution*, implying that managing evolution requires the synchronization between different layers in the software development process. LMP is used as a development framework in which to express and enforce this synchronization process. All examples of LMP discussed in the previous subsection, in some way or another, fit this research theme of co-evolution.

This dissertation can be seen as a first step towards solving the problem of co-evolution of software architecture and implementation. In this context, support for co-evolution boils down to keeping an implementation synchronized with (i.e., ensuring conformance with) its architecture, when either of them evolves. When the software architecture evolves, we are interested in assessing the impact of this evolution on the implementation artifacts. Conversely, we are interested in the impact on the architecture when these implementation artifacts themselves evolve. In other words, we need a formalism that allows automated reasoning about the repercussions at an architectural level of evolving implementation artifacts and about the repercussions of evolving an architecture on these implementation artifacts.

To some extent, conformance checking provides such a formalism. If, after evolution of either the architecture or the implementation, the implementation no longer conforms to the architecture, we know that an evolution problem has occurred. Furthermore, the conformance checking algorithm will give an indication of why and where conformance is invalidated. To avoid first having to apply the evolution, and then re-checking conformance on the entire implementation and architecture, a more incremental solution may again be preferable.

2.3 Software classification

In developing the architecture language proposed in this dissertation, we were inspired by K. De Hondt's research on 'software classification' as an approach to architectural recovery in evolving object-oriented systems [12]. In particular, we experienced the power of the notion of 'virtual software classifications' to codify and reason about software architectures [49].

2.3.1 Traditional software classifications

Classification is a central idea in the object-oriented programming paradigm. Consider for example the Smalltalk language: methods and instance variables are grouped in classes, objects are instances of a class, classes are instances of a meta class, classes belong to inheritance hierarchies, methods are grouped in method protocols, classes are classified in class categories, changes to the Smalltalk image are grouped in change sets, and so on. All of these can be considered as a kind of predefined software classifications. Enhancements of the Smalltalk language, such as the *Envy/Developer*^{TM} version management system, extend the classification possibilities even further (e.g., Envy contains a notion of versions and applications).

2.3.2 The software classification model

In his Ph.D. dissertation, K. De Hondt [12] presents the 'software classification model' as a powerful model to organize implementation artifacts in a flexible and uniform manner. He defines a *software classification* as a simple collection of implementation artifacts, where artifacts can be classified in multiple classifications. For example, in Smalltalk, a software classification could group a set of related classes. All artifacts in a software classification typically share some important characteristic. For example, in a financial application it could be interesting to group all implementation artifacts dealing with "handling deposits" together in a single classification. De Hondt uses these software classifications to capture architectural abstractions that are reverse engineered from implementation artifacts and their dependencies.

As a special kind of software classifications, De Hondt defines a notion of 'virtual software classifications'. Such classifications are not a mere enumeration of implementation artifacts, but are directly extracted from the traditional software classifications, such as inheritance hierarchies and class categories, that can be found in the programming language and development environment (see 2.3.1). These classifications are 'virtual' in the sense that they are actually 'computed' by the environment. When changes are made to the implementation repository, these 'virtual classifications' are automatically recomputed.

In this dissertation, we adopt a slightly broader definition of *virtual software classification*: a virtual classification is a software classification that is specified 'intentionally' (i.e., by 'computing' its elements), as opposed to extensionally. Such virtual classifications are clearly more flexible than ordinary classifications, because they actually 'describe' which artifacts are intended to belong to the classification, instead of explicitly enumerating them. Furthermore, when declared in a logic language, due to the expressive power of that medium, their definitions are often very intuitive and concise, and can be used in multiple ways (e.g., checking, generating, etc.).

Instead of using the terminology of 'virtual classification', a better choice of terminology may be 'computed classification' or 'intentional classification'. However, because it is not our original terminology and because the terminology has already been used in several publications [12, 13, 52, 49], we prefer not to alter it.

2.3.3 The classification browser

To support the creation, manipulation and browsing of software classifications, K. De Hondt developed the *Classification Browser* [12]. It resembles a standard Smalltalk class browser, but with additional features for manually constructing software classifications:

- creating and deleting classifications;
- classifying artifacts in classifications, moving and copying artifacts between classifications, and removing artifacts from classifications;
- support for nested classifications;
- advanced browsing facilities to navigate through the classifications and to navigate through the source code in search for artifacts to be classified;
- predefined virtual classifications (e.g., class categories in Smalltalk).

2.3.4 Virtual classifications

In this dissertation we use virtual classifications of implementation artifacts and the relationships among such artifacts (called 'virtual dependencies'[2]) to declare architectural knowledge explicitly at a sufficiently abstract level, while retaining the ability to perform automated conformance checking.

The main advantage of virtual classifications over explicit enumerations of implementation artifacts is their intentional character.[3] First of all, an intentional definition often has a much more concise representation. Secondly, an extensional definition is less intuitive than an intentional one, which defines precisely which property all entities in the set have in common. For the same reason, the extensional definition is less precise than the intentional one. For example, two classifications can have the same extension, but a different intention. The converse is not true: two classifications that have the same intention, must always have the same extension.[4] Finally, intentional definitions are more robust towards changes than extensional definitions. This is because intentions are true by definition, whereas extensions can be falsified by changing events.[5]

Because of all these advantages, we prefer to use virtual classifications over explicitly enumerated software classifications. The only disadvantage of this choice has to do with efficiency of computation. With an extensional definition, all values are stored explicitly, and thus can be retrieved immediately. An intentional definition can be stored much more concisely, but when its values are needed, they need to be computed from the definition, which may take some time (unless a caching mechanism can be used).

[2] The terminology of 'virtual dependency' was chosen by analogy with the term 'virtual classification'. The dependency is 'virtual' in the sense that it is not necessarily directly visible in the source code, but may require some complex computation to extract it from the source code. It is specified declaratively as a logic predicate over the implementation.

[3] In natural language, the *intention* of a word is that part of meaning that follows from general principles in semantic memory. The *extension* of a word is the set of all existing things to which the word applies. The intention of 'mammal', for example, is a definition, such as "warm-blooded animal, vertebrate, having hair and secreting milk for nourishing its young"; the extension is the set of all mammals in the world [75]. Similarly, in set or type theory, the extension is the collection of all values belonging to that set or type. The intention is a formal definition of these values in terms of some property they all have in common.

[4] Let us illustrate this again with a natural language example taken from [75]. Since 'grandfather' and 'father of parent' have the same intention, they must apply to exactly the same people. On the other hand, 'featherless biped' and 'animal with speech' have the same extension, the set of human beings; but they have different intentions.

[5] Plucking a chicken results in a featherless biped that cannot speak.

2.4 Separation of concerns

Although the main goal of this dissertation is to develop a formalism for checking architectural conformance, as a side-contribution we want to show the relevance and importance of the ideas of separation of concerns at the architectural level. More precisely, we want to illustrate that multiple, potentially overlapping, cross-cutting architectural views may provide a better insight in the overall structure, organization and functionalities of a software system than one single architecture, which is often strongly biased to the structure of the application.

2.4.1 Techniques for separating concerns

W. Hürsch and C. Lopes [32] identified and analyzed the emerging paradigm of *separation of concerns* in software engineering, which tries to formally separate the basic algorithm from special-purpose concerns, such as concurrency, distribution, persistency, and so on. Separating these different concerns, both at a conceptual and at the implementation level, makes the software easier to write, understand, reuse and modify. Examples of techniques and approaches that address the need for separation of concerns are: subject-oriented programming [30], composition filters [2], adaptive programming [39], aspect-oriented programming [36] and hypermodules [63, 80].

For example, aspect-oriented programming (AOP) tries to solve the problem that a program is typically structured according to its base functionality, and that adding 'aspects' that address concerns which cut across this structure typically requires changes throughout the entire program. This problem is caused by what P. Tarr and H. Ossher [63, 80] call the "tyranny of the dominant decomposition". Typically, a software system is decomposed according to one 'dominant' concern and other concerns that cut across this basic functionality are difficult to incorporate in the software. In AOP, there is no dominant concern. The base program and several aspect programs are all implemented separately and are then 'weaved' into one single executable program.

In the same spirit, P. Tarr and H. Ossher suggest to adopt a software development approach which allows a simultaneous decomposition according to multiple, potentially overlapping concerns. They present a uniform model of 'multi-dimensional separation of concerns' to achieve separation of concerns at all levels of the software life-cycle. Most of the techniques for separating concerns mentioned above, can be considered as special cases of this model.

2.4.2 Multiple cross-cutting architectural views

In a recent position paper [48], we made a case for the relevance of the ideas of multi-dimensional separation of concerns at the architectural level. Just like separation of concerns at the implementation level can make source code easier to write, understand, reuse and modify; we claim that multiple, potentially overlapping, cross-cutting architectural views, can provide similar benefits at the architectural level.

Need for multiple cross-cutting architectural views

When designing a building, architects do not make one single plan that describes the overall structure of the entire building. Instead, they use many different plans that each focus on a single aspect or view of the building: front and side views, floor plans, cross sections, foundation, drainage system, electric wiring, central heating, and so on. Not only do these plans address different concerns, they are also supposed to be used by different persons: future inhabitants, bricklayers, electricians, plumbers, and so on. Many of these plans are clearly cross-cutting. For example, a client's request to add an extra window (based on a side view of the building) may require parts of the electric wiring to be reconfigured, since the wiring is often incorporated in the walls. It may even require a partial restructuring of the building, because a window is not a load-bearing structure. It is the architect's job to try and construct a building that optimally satisfies the different constraints and requirements imposed by all these plans.

In contrast with this accepted approach in building architecture, current approaches in the domain of software architecture [64, 77] often assume that software architectures have a direct mapping of the architectural elements to source-code, design-level or physical artifacts and their dependencies. We will refer to these kinds of architectures as *application architectures* because they focus on the actual implementation structure of a software application. For example, in software systems that need to deal with dynamic evolution or runtime reconfiguration [64], the application architecture simply describes what the implementation components are and how they are related to each other. Another example of an application architecture is the component model in UML, which shows the dependencies between parts of the code [77].

Although such application architectures provide good insights into the structure of a software system and thus facilitate the detailed design and implementation as well as the evolution and maintenance of the system, in general there is no need for a software architecture to resemble the application structure itself. The building blocks of a software architecture are merely (abstract) concepts that are meaningful for the application domain. An architecture is a set of relations (or structure) over such concepts. Therefore, in addition to the application architecture, many other kinds of architectures are imaginable and useful; for example, a data-flow or control-flow architecture, or an architecture focusing on a specific concern of the system such as user interaction or distribution, or even architectures addressing domain-specific concerns such as rule-based interpretation (in the domain of rule-based systems). Many of these architectures, however, often 'cut across' the implementation structure or application architecture.

An important side-contribution of this dissertation is to illustrate the need for multiple, potentially overlapping, cross-cutting architectural views. The idea that a software system can have not only an application architecture, but also many other architectural views that address specific concerns, is slowly infiltrating in the software architecture community [7]. P. Kruchten [38] proposes his '4 + 1 View Model' which describes a software architecture using five concurrent views. We agree that multiple architectural views are useful, but do not restrict an architect to a fixed number (i.e., 5) of predefined views. The architect should be able to use as many architectural views as needed, and should not be restricted to a predefined set of concerns that these views should address.

Checking conformance to multiple cross-cutting architectural views

The case study which will be described in Chapters 4 and 7 is the continuation of an experiment reported on in an earlier paper [52].[6] In that paper, we tried to check conformance of the Smalltalk implementation of the SOUL language to the typical architecture of a rule-based interpreter [74]. The elements in this architectural view did not always map straightforwardly to the classes or other implementation artifacts. For example, the 'Rule Interpreter' component at architectural level corresponded to many different methods implemented by many classes in the entire implementation. Nevertheless, by defining a cross-cutting architectural mapping we were still able to check architectural conformance. This initial experiment made us realize that an architecture which provides a high-level view of some aspect of the design of a software system, does not necessarily need to have a direct mapping to the implementation, but may cut across it.

The need for multiple cross-cutting views that need to be kept consistent with an implementation is also recognized by current research on traceability. For example, P. Garg and W. Scacchi [24] propose the use of a hypertext system to manage software life-cycle documents, including architectural designs. Keeping documents consistent, complete and traceable is seen as a critical problem, especially for large software systems. All documentation, as well as program code, is expressed as hypertext nodes that are linked together. Such a hypertext approach naturally supports cross-cutting links among documents. The proposed hypertext system focuses mainly on the editing, structuring, and browsing of documents. Although the system does provide some hooks to incorporate consistency checking tools (e.g., through automated links and by offering a uniform

[6] The formalism proposed in this dissertation is a refinement of the formalism proposed in [52]. Amongst others, the architecture language used in this dissertation adds a notion of ports and roles.

tool interface), Garg and Scacchi do not specify in detail how conformance across documents is to be achieved. Therefore, our approach is complementary to theirs.

Terminology

Based on the insight that architectural views do not necessarily require a direct mapping of their architectural entities to implementation or physical components, but that this architectural mapping may cross-cut the implementation, we are *not* tempted to follow Shaw and Garlan's [74] example to speak about 'components' at an architectural level. Although many definitions of (software) components exist, most of them seem to agree at least on the fact that a software component is a (reusable and replaceable) piece of implementation of a software system. For example, UML defines a component as a "distributable *piece of implementation* of a system, including software code (source, binary or executable) but also including business documents" [77, 60]. Broy defines a component as "a *physical encapsulation* of related services according to a published specification" [8]. Other definitions of the term, mentioned in [77], are: "a *physical* and replaceable *part of a system* that conforms to and provides the realization of a set of interfaces; "an *executable software module* with identity and a well-defined interface"; "an encapsulated *part of a software system* with an interface that provides access to its services"; "an *object* that lives in the *binary* world", and so on. Many more similar definitions of the term can be found in research literature [9, 79].

Most authors [4, 33, 37, 64, 74] consider software architecture merely as a structural description of the interaction among the software components of which the system is constructed. In this view, there is no objection against using the term 'component' at the architectural level. Extending the usage of the term 'component', however, to represent architectural entities that may correspond to many artifacts spread throughout the entire implementation, does not seem to be a good idea. The above definitions indicate that the term 'component' has the connotation that it corresponds to some localized implementation artifact, and extending the definition to allow components that cut across the implementation would only give rise to confusion. We prefer to use the term (architectural) 'concept' to denote architectural entities. This corresponds to our intuition that a software architecture expresses relations (or structure) over abstract concepts that have some meaning for the application domain. How these concepts are actually implemented is not important at this level of abstraction. So instead of talking about architectural components and connectors (as, for example, in [74]), in the remainder of this dissertation we will talk about *architectural concepts* and *architectural relations* respectively.[7]

To keep the reader from getting lost in all the new terminology that is introduced in this dissertation, Appendix C contains a small thesaurus of terminology that is specific to this dissertation.

[7] The chosen terminology is inspired by and consistent with the terminology used in the research domains of knowledge representation and ontologies. J. Sowa [75] proposes the theory of conceptual graphs as a method of representing mental models of some problem domain. Such graphs consist of *concepts* and *conceptual relations* between these concepts. According to M. Uschold and M. Gruninger [83], an ontology embodies some sort of world view with respect to a given domain. The world view is often conceived as a set of *concepts*, their definitions and their *relationships*.

Chapter 3

Problem Statement

In this dissertation, we tackle the research problem of checking that the implementation of a software system conforms to its architectural views. More particularly, we want to develop an expressive formalism for architectural conformance checking that is easy to automate in tools. A logic meta-programming medium suggested as an obvious candidate in which to express this formalism.

3.1 Automating architectural conformance checking

The research topic of *software architecture* is gaining more and more importance, and is slowly being adopted by industry [4]. It is reasonably well understood how software architectures can be used for forward-engineering purposes, i.e., designing and implementing a system in accordance with a certain software architecture. However, once an architecture has been designed and a system based on this architecture has been implemented, the architecture enters a maintenance phase. If no precautions are taken to ensure that the implementation remains conform to this architecture, the architecture quickly becomes outdated due to the problems of architectural erosion and architectural drift. This leads to a vicious circle, where modifications are only made to the implementation, because the architecture is not up to date anyway, which causes the implementation to drift away even further from the architecture. This seems to be one of the main reasons why, in current-day practice, software architectures are not used to their full potential. Therefore, adequate techniques are needed to keep the architecture consistent with the source code. Architectural conformance checking is such a technique.

In this dissertation, we tackle the research problem of developing a formalism for architectural conformance checking that can easily be automated and incorporated in tools. In particular, we focus on the problem of verifying whether the implementation of a software system corresponds to the high-level structure prescribed by the software architecture. We do not take run-time information into account and restrict ourselves to the static structure of the implementation only.

Expressiveness was a major driving force in our research. Although a few conformance checking techniques already exist, these were essentially developed from the viewpoint of efficiency. However, these techniques do not seem to be sufficiently general or expressive to be used in practice. Therefore, we approach the problem from the opposite direction, and develop a conformance checking formalism that is as expressive as possible, even if this implies a loss of efficiency. For this purpose, a logic meta-programming approach (see Section 2.2) is adopted. It should be stressed here that LMP is more than merely a suitable implementation medium for the conformance checking algorithm. By explicitly declaring the architectural mapping in terms of virtual classifications and virtual dependencies, which can make use of the full power of the LMP language, a very expressive formalism is obtained. It allows an architect to declare very complex architectural mappings in a reasonably intuitive and concise way.

As explained in Section 2.1, much current research on software architectures assumes a direct mapping of the architectural entities to physical or implementation-level entities. We agree that architectural views with such a direct mapping to a system's implementation can be important software engineering assets. As explained in Subsection 2.4.2, however, we claim that it is equally important to consider architectural views with a less straightforward architectural mapping. For example, architectural entities may correspond to implementation artifacts spread throughout the entire implementation. Therefore, our conformance checking technique should provide support for expressing (and checking conformance to) multiple (cross-cutting and potentially overlapping) architectural views.

To summarize, the goal of this dissertation is to provide an expressive formalism, based on LMP, to reason about conformance of the implementation of a software system to one or more architectural views. The formalism should be as intuitive and simple as possible, so that it is easy to incorporate in tools, and so that it will be accepted by software engineers. The notions of virtual classifications and virtual dependencies will be put forward as simple, yet very powerful, mechanisms for abstracting implementation artifacts and their dependencies into architectural concepts and relations.

3.2 Novelty of the approach

We mentioned some research results on architectural conformance checking in Subsection 2.1.5. In our opinion, however, most of these results do not provide a sufficiently expressive technique to check architectural conformance. To justify this claim, we discuss each of these techniques below, and explain why they lack expressiveness. Based on this discussion, we compile a list of desiderata for a sufficiently expressive formalism for automated architectural conformance checking. This strong focus on expressiveness is an important contribution of our work. We conclude with a justification of why LMP is an ideal approach for implementing such an expressive formalism.

3.2.1 Existing conformance checking approaches

ADLs

We already mentioned in Subsection 2.1.5 that architectural conformance checking is one of the areas in which existing ADLs are most lacking. Most ADLs do not provide support for refining an architecture to the implementation of a system, nor for traceability of changes across levels of refinement [45]. Only the ADLs 'SADL' [56] and 'Rapide' [43] support refinement and traceability to a certain extent. Both provide refinement maps for architectures at different abstraction levels, but do not support refinement to the implementation level.

MILs

Module interconnection languages [68] were a first step towards current-day ADLs, and enabled the formal description of the global structure of a software system in terms of its modules and their interconnections. These descriptions could be processed automatically to verify system integrity. According to Shaw and Garlan [74], a problem with MILs is that they force software architects to use a lower level of abstraction than is appropriate, because they focus too much on 'implementation' rather than on 'interaction' relationships between modules.

Software reflexion models

An approach that is closely related to ours is that of 'software reflexion models' [57]. In this approach, an engineer defines some 'high-level model' of the software using boxes and arrows, extracts a 'source model' (such as a call graph or an inheritance hierarchy) from the source code, and defines a declarative mapping between these two models. Using this information, a 'software reflexion model' is computed, which summarizes the main correspondences and differences between the high-level model and the source model. Our approach is similar in spirit: we declare some high-level architectural view, define a declarative mapping of the architectural entities to source code artifacts and their dependencies, and compare the source code to this architectural view.

The main differences between both approaches have to do with expressiveness. Whereas the software reflexion-model approach stresses efficiency, our approach is situated at the other end of the spectrum. We consider expressiveness as a major criterion in the development of our conformance checking formalism. Therefore, we adopt a LMP approach which combines the expressive power of multi-way querying, logic inferencing and unification. Our goal is to allow describing software architectures, and their mapping to the implementation, at the highest abstraction level possible, without losing the ability to verify architectural conformance of source code.

Now, let us take a closer look at some of the differences between both approaches:

- The software reflexion-model approach typically maps high-level model entities to physical (e.g., modules, directories or files) or logical (e.g., classes or functions) source-model entities. In our approach, architectural concepts can be mapped to any (collection of) implementation artifact(s). We will often use mappings of architectural concepts to multiple implementation artifacts spread throughout the source code (*cross-cutting mappings*), or to a mixture of different kinds of implementation artifacts (*heterogeneous mappings*).

- In the software reflexion-model approach, the arrows in the high-level model have no associated semantics. They are merely compared syntactically with the arrows in the source model, which were extracted from source-code dependencies. As the high-level model cannot distinguish between different kinds of arrows, the source models typically consider one kind of dependencies only. Our approach is a more semantic one. A single architectural view may contain many different kinds of architectural relations. Similar as for architectural concepts, we define an explicit mapping of architectural relations to implementation dependencies.

- The kinds of relationships that are typically considered in the software reflexion-model approach are implementation dependencies such as calling relations, file or data dependencies, cross-reference lists, inheritance hierarchies, and so on. In other words, just like MILs, software reflexion models focus essentially on 'implementation' rather than on 'interaction' relationships among the high-level model entities. In our approach, we can use LMP to describe arbitrary complex relationships dealing with transitive closures, interaction and collaboration protocols, programming conventions, design patterns, and so on.

- The declarative mapping in the software reflexion-model approach uses (efficient) regular expressions to extract patterns of interest from the source-code. To obtain a maximum of expressiveness, our architectural mapping uses the full power of LMP. To define the mapping we can use a combination of techniques such as string pattern matching, logic reasoning about method parse trees, semantic inferencing, and so on.

- Both approaches allow multiple source-code entities to be mapped to the same high-level model entity, and vice versa.

Gestalt

Another approach that was mentioned in Subsection 2.1.5 was 'Gestalt' [72]. The Gestalt toolset can check structural consistency between the architecture and the code. Gestalt acknowledges the need for providing 'implementation' as well as 'interaction' relationships. Furthermore, it allows for composite architectural concepts and relations, i.e., concepts and relations that are described in terms of other architectural concepts and relations;

Gestalt does not support cross-cutting mappings from an architecture to the implementation. In Gestalt, 'consistency' means structural, or topological, consistency. Intuitively, this means that the code should be broken into parts that correspond to the parts of the architecture, and that the paths of communication between parts of the code should correspond to the paths of communication specified in the architecture.

Conclusion

Although a few architectural conformance checking approaches exist, we did not find any approach with all of the following features:

explicit concept mappings that map architectural concepts to one or more implementation artifacts; including:

  cross-cutting mappings of architectural concepts to multiple implementation artifacts spread throughout the source code, for example, a group of related methods belonging to many different classes spread throughout the entire implementation;

  heterogeneous mappings of architectural concepts to groups of implementation artifacts consisting of a mixture of different kinds of artifacts (like classes, methods and variables);

  composite architectural concepts that are described in terms of other high-level concepts and relations;

explicit relation mappings that map architectural relations to implementation dependencies, including:

complex architectural relations that deal with transitive closures, interaction and collaboration protocols, programming conventions, design patterns, and so on;

composite architectural relations that are described in terms of other high-level concepts and relations;

As we will see, our formalism does have these features.

We repeat that our approach focuses mainly on expressiveness, perhaps at the cost of decreased efficiency. Efficiency, however, may not be so crucial for tools that check architectural conformance. There is no real need to check architectural conformance in 'real time'. We can always run such a check in background, or overnight, and inspect the results later. Furthermore, efficiency strongly depends on the available technology. Machines are always getting faster, so that algorithms that are considered too slow today, may be considered fast enough tomorrow. Nevertheless, we will still try to make our prototype conformance checking tool as efficient as possible. We will even discuss some optimizations to increase the efficiency of the proposed conformance checking algorithm. For example, we will sketch how the algorithm could be turned into a more efficient incremental version.

3.2.2 Criteria for our architectural formalism

Below we compiled a list of criteria that our formalism for checking architectural conformance should satisfy. Each criterion is subdivided in a non-exhaustive list of requirements that are relevant to that criterion.

Expressiveness. In order for the formalism to be sufficiently expressive, it should:

1. pose no a priori restriction on the kinds of implementation and architectural entities and relationships that can be considered;
2. allow for composite architectural concepts and relations that are defined in terms of a sub-architecture;
3. allow for complex architectural relations that can deal with transitive closures, interaction and collaboration protocols, programming conventions, design patterns, etc.;
4. allow for cross-cutting mappings of architectural concepts to implementation artifacts: one architectural concept may correspond to multiple artifacts spread throughout the implementation;
5. support the definition of multiple, potentially overlapping, architectural views on the same software system.

Simplicity. The formalism should be as simple and intuitive as possible, so that it can:

1. easily be incorporated in tools;
2. easily be understood and used by architects and software developers.

Extensibility. The formalism should be flexible and expressive enough, so that it

1. is customizable with predefined and user-defined architectural abstractions;
2. can easily be extended to deal with, for example, architectural patterns and architectural styles.

Generality. The formalism should be sufficiently general so that it can also be used to deal with architectural conformance checking of, for example:

1. implementations in other object-oriented programming languages;

2. implementations in other programming paradigms;
3. design models.

The requirements enumerated in the criterion of expressiveness were essentially extracted from the discussion in Subsection 3.2.1. An extra requirement was added to address the need for allowing multiple architectural views on the same software system. The criterion of efficiency (both run-time and memory efficiency) was deliberately *not* included in the above list of criteria. As explained in Subsection 3.2.1, we are willing to sacrifice efficiency in favor of increased expressiveness. The criterion of simplicity captures our preference of building a formalism that is intuitive and easy to understand by software engineers, and easy to incorporate in tools. Finally, the formalism should be sufficiently generalizable and extensible to new domains and with new features, so that it can be used as the basis for building an industrial-strength tool.

It is not our intention to actually build a tool that includes all features mentioned in the above list. Nevertheless, the formalism we will propose should be powerful enough so that it enables the construction of such a tool. To prove the feasibility of building such a tool, we will actually implement a prototype which does satisfy most of the above criteria. For those features that are not implemented, we will explain how the prototype could be extended or generalized to deal with them.

3.2.3 Logic meta programming

To conclude this section, we explain why a LMP approach seems like an ideal choice to define an architectural model and conformance checking algorithm that satisfies the above criteria. Not only is a logic meta language a suitable medium in which to implement our conformance checking algorithm, we also use it as an expressive medium in which an architect can define the architectural mapping of architectural concepts and relations to implementation artifacts and their dependencies.

There are three good reasons for choosing a LMP language:

1. A logic language is typically well suited for representing and declaring knowledge. In fact, our architectural model naturally grew out of some experiments we conducted to declaratively codify the conceptual structure of a software system at a high level of abstraction [52]. Oreizy [62] reports that most (dynamic) architecture description languages use declarative descriptions, to facilitate static analysis of the descriptions.

2. Logic programming is an expressive medium for reasoning about (architectural) knowledge, thanks to its declarative nature, its expressive power, its capacity to support multi-way queries, and the powerful built-in techniques of unification and backtracking.

3. Logic programming has long been identified as very suited to meta programming and language processing in general. We use the logic language as a meta language to reason at an architectural level about the implementation artifacts and their dependencies in an object-oriented base language.

The implementation of the conformance checking algorithm uses LMP to reason about the architectural descriptions and their architectural mapping. More precisely, it checks whether the implementation conforms to the more abstract structure prescribed by the architectural descriptions. The conformance checking algorithm can naturally and straightforwardly be implemented in terms of the construction and evaluation of some logical expression [50]. In short, this is achieved as follows. The architectural descriptions are translated into a logical expression. In this translation, architectural relations are replaced by high-level implementation relationships and architectural concepts are replaced by the (groups of) concrete implementation artifacts which they represent. Evaluating this resulting logical expression corresponds to checking conformance of the implementation to its architecture.

The architectural mapping of architectural entities to implementation artifacts and dependencies uses LMP to declare the conformance between the architecture and the implementation. The mapping for architectural concepts is defined in terms of 'virtual classifications' and the mapping for architectural relations is defined in terms of 'virtual dependencies'. By providing the full power of the logic meta language to define these virtual classifications and virtual dependencies, we obtain a maximum of expressiveness, and can easily satisfy all requirements enumerated in the criterion of expressiveness, as will be elaborately explained in Chapter 7.

3.3 Validation

To validate our thesis, we proceed in three steps. First, we develop a language and associated algorithm for checking conformance of the implementation of a software system to one or more architectural views. Then we use LMP to implement a prototype of a conformance checking tool based on this language and algorithm. Finally, we perform a concrete case study, using the implemented prototype. In addition to proving the feasibility of the developed formalism, the case study illustrates the expressiveness of using a LMP approach to define the mapping from architecture to implementation.

3.3.1 The formalism

The proposed formalism for architectural conformance checking will be explained in three parts:

1. the architecture language which consists of an ADL and an architectural mapping language;
2. the conformance checking algorithm;
3. the actual implementation of a prototype conformance checker based on this language and algorithm.

Both the architectural mapping language and the conformance checking algorithm have a strong logic flavor and LMP is used to implement the prototype. LMP is more than a convenient implementation medium, however. To define complex architectural mappings, the architect can also make full use of the expressive power of LMP.

Taking inspiration from K. De Hondt's [12] positive experiences with recovering design knowledge in terms of simple software classifications (see Section 2.3), we decided to use software classifications as an intuitive but expressive abstraction mechanism for mapping architectural concepts to sets of implementation artifacts. In particular, we focus on 'virtual classifications' which intentionally describe their elements. This enables an architect to express a concept mapping in an elegant and concise way, instead of explicitly having to enumerate the implementation artifacts to which the concept corresponds.

The architectural relations will be mapped to 'virtual dependencies', which are logic predicates describing high-level implementation or design relationships among implementation artifacts. Again, this gives an architect considerably more expressive power than when he would be restricted to using a fixed set of predefined relation mappings.

3.3.2 The case study

The case we studied is the architecture and implementation of the SOUL system. It is a well-designed, medium-sized application containing about 100 Smalltalk classes. In dialogue with the main SOUL developer, three different architectural views on SOUL were defined, and conformance of its implementation to these architectural views was checked.

Apart from illustrating the conformance checking algorithm, the case study proves the expressiveness of our approach. This expressiveness is mainly due to the powerful combination of LMP with the abstraction mechanisms of virtual classifications and virtual dependencies. Based on the results of the case study, we show that our formalism satisfies most criteria put forward in Subsection 3.2.2, and the criterion of expressiveness in particular. We will provide examples on how to express multiple architectural views, complex architectural relations, cross-cutting mappings, composite concepts and relations, and so on.

The case study is the topic of the next chapter.

Chapter 4

Case: The Architecture of SOUL

The case study used throughout this dissertation is the architecture and implementation of the SOUL system. We introduce SOUL and present a number of different architectural views on the SOUL implementation. Each of these views focuses on a different concern of the system.

4.1 The Smalltalk Open Unification Language

4.1.1 The SOUL system

The SOUL system was developed by R. Wuyts at the Programming Technology Lab of the Vrije Universiteit Brussel [86]. The *Smalltalk Open Unification Language* (SOUL), is a logic language that allows intelligent querying and meta-level reasoning about Smalltalk code. The syntax of the language is similar to that of the logic programming language Prolog, but has an extension that allows logic clauses to manipulate elements from the Smalltalk language (such as classes, inheritance relations, method bodies). It is even possible to execute blocks of Smalltalk code (that may refer to instantiated logic variables) during the interpretation of queries. This strong symbiosis between a logic language and Smalltalk was made possible by implementing SOUL entirely in Smalltalk and by using the powerful reflective capabilities of the Smalltalk environment. Currently this symbiosis is being extended even further to allow logic queries to be triggered transparently from within Smalltalk source code.

SOUL includes a declarative framework of rules that allows reasoning about Smalltalk programs at the implementation, design and architectural level [52, 86]. This framework has a layered structure. The lowest layer is a Smalltalk-specific layer which defines some primitive predicates for manipulating Smalltalk source-code artifacts and implementation relationships. On top of this layer resides a layer that adds some structural predicates defined directly in terms of the more primitive predicates. Higher-level layers describe more high-level relationships, such as design patterns and architectural constraints.

The SOUL-Smalltalk combination has proven to be an ideal medium for building sophisticated software engineering tools. Amongst others, experiments have been carried out to support best-practice patterns, idioms, and coding conventions [54]; to build sophisticated 'find and replace' tools; to detect and check design patterns in Smalltalk source code [86]; to log violations of certain programming conventions and styles in a 'to do' list; to separate the aspect[1] of domain knowledge from the implementation aspect [18, 17]; etc. In the context of this dissertation, we used SOUL both as a case and as a medium to check conformance of Smalltalk source code to architectural descriptions.

[1] in the sense of aspect-oriented programming

4.1.2 Architectural views

In the following sections we describe the architecture of the Smalltalk implementation of SOUL, as seen from three different points of view. Each of these architectural views focuses on a different aspect of the SOUL system. In this chapter we merely introduce and describe these different views. In later chapters we show how they are mapped to the implementation.

In the 'user interaction' architectural view (see 4.2.2) we concentrate on those concerns of the system that are important for a user of the system, i.e., how a user can interact with the system. In the 'rule-based interpreter' view (see 4.3), we focus on the core functionalities of the system, which are related to the interpretation of logic queries and rules. Although the architecture of a rule-based system has been well-documented in literature on software architecture [4, 31, 74], we refine this architecture somewhat to stress some important features of the SOUL system. A final architectural view we describe is the actual 'application architecture' of SOUL (see 4.4.2), which explicitly describes the high-level structure of the implementation of the SOUL system.

All these different (and partially overlapping) architectural views contribute to the understanding of the SOUL software system as a whole, by highlighting certain aspects of it. Two of these views cross-cut the actual implementation structure. Only one view, the 'application architecture', maps directly to the chosen implementation decomposition.

4.1.3 Notational conventions

The following notational conventions are used when discussing the different architectural views: in running text, architectural concepts are printed in **bold** and architectural relations in *italic*. We also present each architectural view graphically, using the simple graphical notation of Figure 4.1.

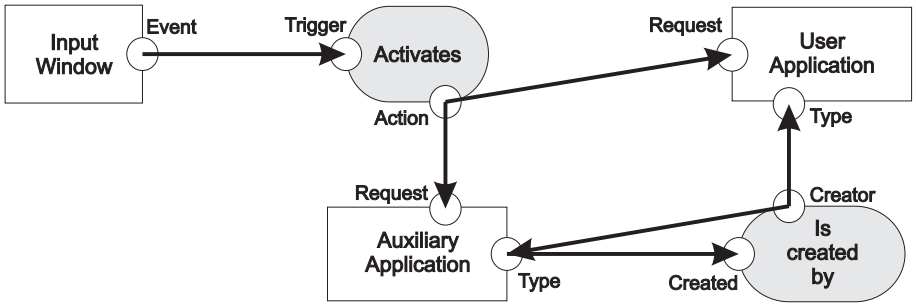

Figure 4.1: Graphical representation of an architectural view.

This graphical notation is somewhat similar to that of 'conceptual graphs' [75], as it shows those concepts that are important for a certain architectural view, as well as the relations between those concepts. Whereas the concepts represent the entities of interest in a certain architectural view, the relations describe how they relate to each other, i.e., which role the entities play. Concepts are depicted as white rectangles and relations as grey round-corner rectangles. Typical examples of architectural concepts for the SOUL system are: **Input Window**, **User Application**, **Auxiliary Application**, **Rule Interpreter** or **Query**. Typical examples of architectural relations are: *Activates*, *Is Created By*, *Asks*, *Uses Data* and *Is Kind Of*.

The ports of an architectural concept, which define how a concept may interact with its environment, as well as the roles of an architectural relation, which identify the participants for that relation, are represented graphically as little circles at the begin and end points of the lines that link concepts and relations. In our formalism, these links are undirected; they just connect a port with a role. Nevertheless, in our pictures we will often put arrows on the links to make the diagrams more readable. More precisely, we adopt the following notational convention: architectural relations are named as verbs; an arrow pointing towards a relation designates the subject of the verb and the arrows pointing away from a relation designate the direct and indirect objects of the

verb. For example, the *Activates* relation has two roles: 'Trigger' and 'Action'. 'Trigger' has an incoming arrow because it is the subject of the activation (i.e., it represents the thing that causes the activation). 'Action' has an outgoing arrow because it is the direct object of the activation (i.e., it represents the thing that is being activated). We stress that this is only a notational convention and that there is no real semantics associated with the arrows, nor is it enforced by the tool.[2]

The exact semantics of our notation will be explained in Chapter 5. In the next section, we discuss our first architectural view on SOUL: the 'user interaction' architectural view. Note that Figure 4.1, which we used to illustrate our graphical notation, is actually a subset of the 'user interaction' architectural view.

[2]We will come back to this notational convention later, after having explained what the semantics of the other architectural entities are.

4.2 User interaction

In the 'user interaction' architectural view we are mainly interested in the interaction of a user with the SOUL system, and the main functionality of the SOUL system from a user point of view.

4.2.1 SOUL applications

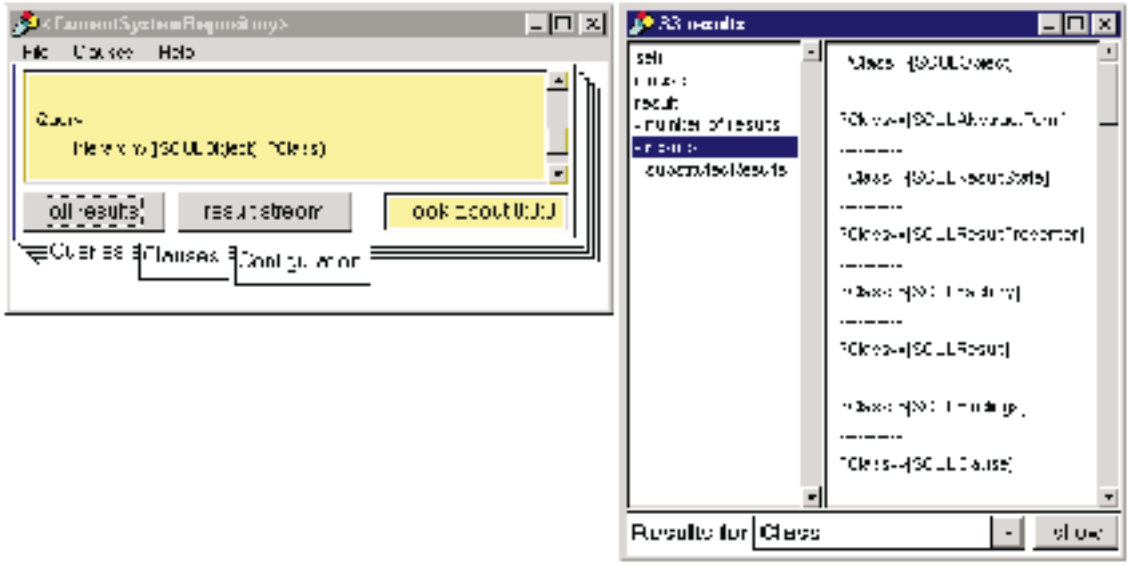

Figure 4.2: The SOUL Query Application.

A typical example of how a user might interact with the SOUL system is depicted in Figure 4.2. The window on the left is the standard input window of the SOUL 'Query Application'. In this window, a user can type a logic query to be interpreted by the SOUL system. An example[3] of such a query is

```
Query hierarchy([SOULObject], ?Class)
```

which can be used to find all direct and indirect subclasses of a given Smalltalk class `SOULObject`. After all results have been computed, an output window such as the one on the right in Figure 4.2 is obtained. For this particular query, 33 results are generated. This kind of usage is similar to how a user would interact with the interpreter of any other logic programming language.

Another way of interacting with the SOUL system, which is easier for non-expert users, is by using the 'Structural Find Application'. This is illustrated in Figure 4.3. This application transparently uses logic queries to allow searching for methods or classes in the Smalltalk image using complex search patterns. The user only needs to fill in one or more simple selection fields and the Find Application will automatically generate and interpret the corresponding query for the user. For example, the Find Application may be used to find all classes that have a name matching some pattern, have a method sending some specified message and implement a method with a given name. An example of an input window for the Find Application is the window on the left in Figure 4.3. The results of executing the Find Application are presented in a more readable form than when the Query Application is used directly, as can be seen by comparing the output window on the right in Figure 4.3 (created by the Find Application) with the one on the right in Figure 4.2 (created by the Query Application). On the other hand, the Query Application is more general and flexible, because queries are not restricted to some fixed set of selection fields,

[3] The complete syntax of the SOUL language can be found in Appendix A.

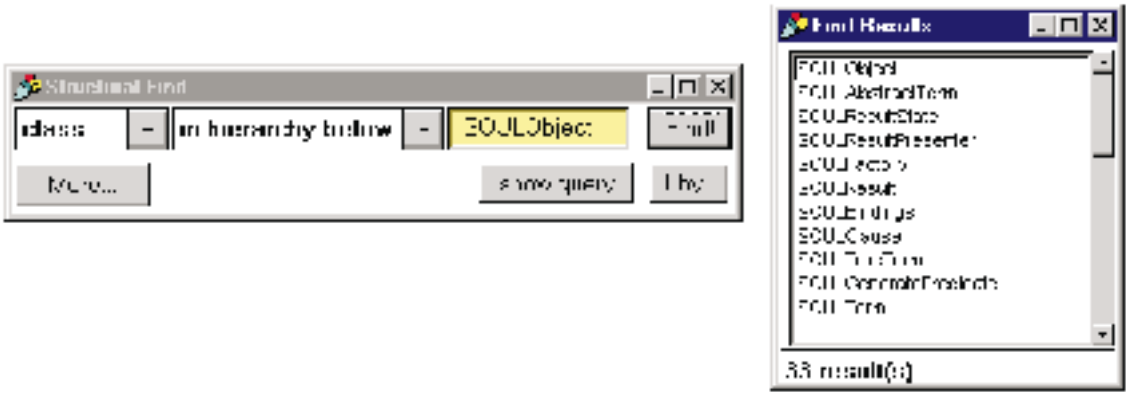

Figure 4.3: The SOUL Structural Find Application.

as in the Find Application. Instead, the full SOUL syntax and all predefined predicates may be used to construct a query.

4.2.2 The user interaction architectural view

In the previous subsection we informally explained how users can interact with SOUL. In this subsection we codify this intuition in terms of an architectural view which focuses on the concern of 'user interaction' with the SOUL system. Figure 4.4 depicts this architectural view. It is centered around a **User Application** architectural concept representing the different kinds of SOUL applications. At this level of abstraction, the specific SOUL applications, i.e., the Structural Find Application and Query Application, are considered as implementation details. They are all represented by the same generic **User Application** concept.

SOUL user applications are typically *activated* as a result of a certain event triggered by a user in an **Input Window**. For example, after typing in a query in the input window of the Query Application, the user presses the "all results" button. This causes a request to compute the results of the user's query, to be sent to the Query Application. Similarly, after filling in the selection fields in the input window of the Structural Find Application, the user presses the "Find!" button. This causes a request to be sent to the Structural Find Application to compute the results of the generated query corresponding to the user's inputs. In general, an 'event' generated by an input window 'triggers' a 'request' on a **User Application** (or one of its **Auxiliary Application**s) to activate a certain 'action'. This is depicted in Figure 4.4 by the *Activates* architectural relation from the **Input Window** architectural concept to the **User Application** and **Auxiliary Application** concepts.

Note that the 'Action' role of the 'Activates' relation has two links attached to it. As we will explain in Section 5.2 and Section 5.4, this should be read as a disjunction, not as a conjunction. An 'Input Window' activates either a 'User Application' or an 'Auxiliary Application'. To represent a conjunction, we would explicitly draw two separate 'Activates' relations: one from 'Input Window' to 'User Application' and one from 'Input Window' to 'Auxiliary Application'.

An **Auxiliary Application** is an application that *is created by* a **User Application** or by another **Auxiliary Application** to do part of its computation. This dependency is modeled through the *Is Created By* architectural relation on Figure 4.4. (Again, we need a disjunction here and modeled this by attaching two outgoing links to the same 'Creator' role.)

The **User Application** concept is linked to the **Query Interpreter** concept by means of an *Asks* architectural relation. This relation represents the fact that a **User Application** typically needs to compute the result of a query. More precisely, the semantics of the *Asks* relation (see later) stipulates that the **User Application** invokes the **Query Interpreter** (to interpret a query). After the result (of this query) has been computed by the **Query Interpreter**, it is

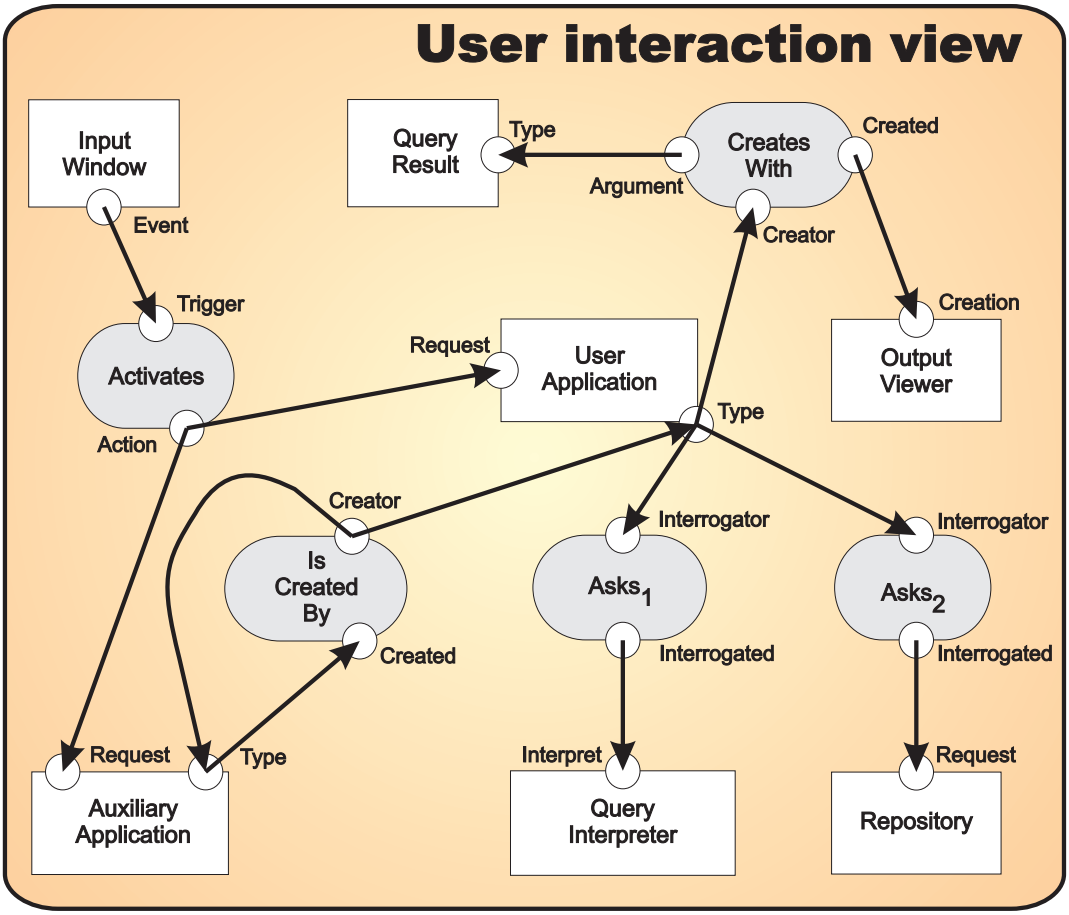

Figure 4.4: SOUL 'user interaction' architectural view.

returned to the **User Application** for further processing.

The **User Application** concept is also linked to the **Repository** concept by means of an *Asks* architectural relation. This models the fact that user applications sometimes need information directly from the fact and rule **Repository** (and not only indirectly via the query interpreter). The retrieved information is then used by the **User Application** for further processing. For example, the Query Application does this to retrieve the available set of clauses in the repository in order to show them to the user. It also allows the user to modify this set (i.e., add or remove clauses) and propagates these modifications to the repository. (Note that although both *Asks* architectural relations in Figure 4.4 have the same semantics, we have given them a different name, i.e. $Asks_1$ and $Asks_2$, so that they can easily be referred to.)

Instead of returning results of queries to the user directly, they are presented in a so-called **Output Viewer** which allows easy browsing and inspecting of these results. After a **User Application** has asked the **Query Interpreter** to compute some **Query Result**[4], the **User Application** will create an **Output Viewer** with this **Query Result**. This is depicted by the *Creates With* architectural relation in Figure 4.4.

This concludes our introduction to the 'user interaction' view of the SOUL system. Now we turn our attention to a second important architectural view of SOUL: the 'rule-based interpreter' view.

[4] Although one might expect the $Asks_1$ architectural relation to be a ternary relation linking **User Application**, **Query Interpreter** and **Query Result**, for now we model it as a binary relation between **User Application** and **Query Interpreter**. In Subsection 8.1.3 we will show how to refine this binary relation into a ternary one.

4.3 Rule-based interpreter

Figure 2.1 on page 20 depicted the basic architecture of a rule-based interpreter. Since the SOUL rule-based system includes a rule-based interpreter, we expect (part of) the implementation of SOUL to be conform to this architecture. To highlight the important features of the SOUL rule-based interpreter, we will further refine the basic architecture of Figure 2.1. For this purpose, we use the more detailed notation of Figure 4.1. Furthermore, as explained in Subsection 2.4.2, we will now talk about architectural concepts and relations, as opposed to architectural components and connectors, respectively.

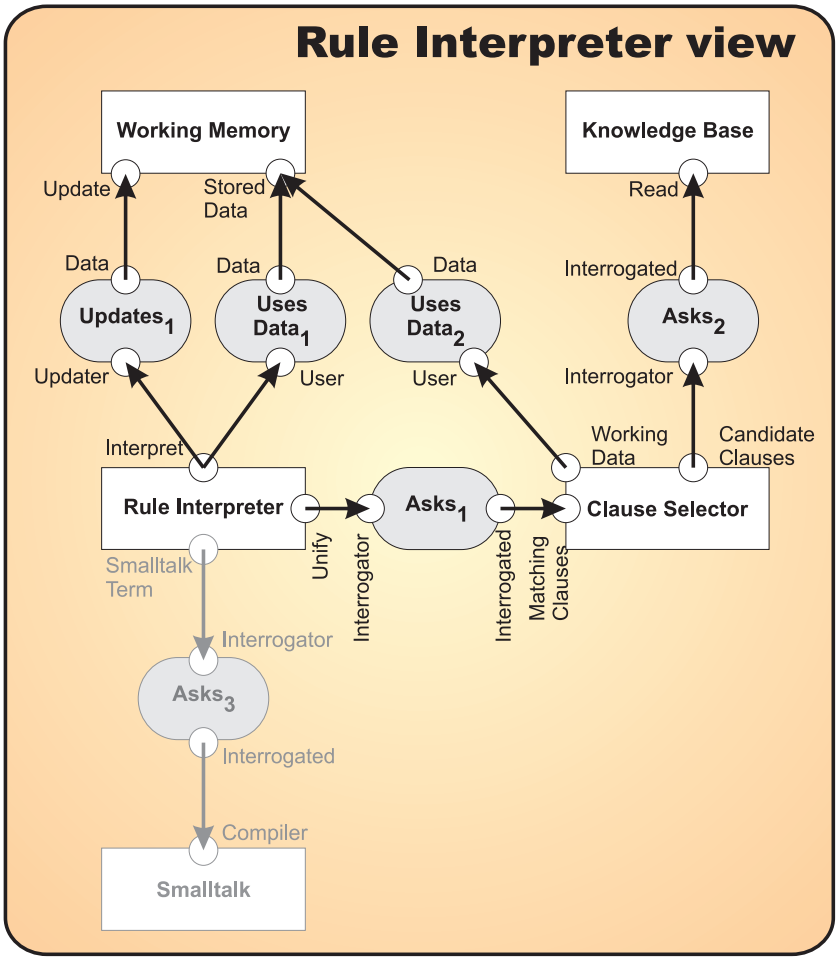

Figure 4.5: SOUL 'rule-based interpreter' architectural view.

The 'rule-based interpreter' architectural view of SOUL is depicted in Figure 4.5. If one ignores the architectural relation between **Rule Interpreter** and **Smalltalk** it is clear that this architecture straightforwardly corresponds to the one depicted in Figure 2.1. One minor difference is that some of the arrows point in the opposite direction. This is due to our notational convention (see 4.1.3) of naming architectural relations as verbs and making the arrows point to the subject of the verb, and away from the direct and indirect objects of the verb. (However, as explained earlier, no real semantics is associated to these arrows.) Another difference is that we left out the 'Inputs' and 'Outputs' links because the 'user interaction' architectural view already elaborated on the input and output interactions with a user of the rule-based system. A third difference is that the $Asks_3$ relation between the **Rule Interpreter** and **Smalltalk** was not present on Figure 2.1. It is specific to the SOUL system because of its close symbiosis with the Smalltalk environment.

Now let us take a closer look at this architectural view. The **Rule Interpreter** concept represents the actual interpretation process. During interpretation, some **Working Memory** is used to store the intermediate results of the interpretation process, i.e., the accumulated bindings of values to logic variables. The facts and rules representing the data and control of a SOUL program are stored in a **Knowledge Base** of logic clauses. During the interpretation process, clauses are looked up (read) in this logic repository, via the **Clause Selector**.

When interpreting a query, it is decomposed into more primitive terms, each of which will be interpreted separately (using the bindings of the already interpreted ones). To interpret a term, it is passed to a **Clause Selector** which has the task of finding all clauses (i.e., facts or rules) in the repository, that have a head which matches the term being interpreted. During this process, the formal parameters of the clauses in the repository are unified with the actual argument values of the term being looked up. These bindings of variables to values are stored in the **Working Memory**. Finally, the matching clauses are returned to the **Rule Interpreter** for further processing.

In addition to the features provided by a typical logic programming language like Prolog, SOUL provides some language primitives — called 'Smalltalk terms' — that enable the execution of blocks of Smalltalk code as part of logic clauses. To interpret these primitives, the **Rule Interpreter** accesses the **Smalltalk** image directly. More specifically, the Smalltalk compiler is invoked to compute the value of these primitives. To increase the expressiveness of the symbiosis between SOUL and Smalltalk even further, it is allowed for 'Smalltalk terms' to refer to logic variables, but only if they are instantiated. Before compilation, these logic variables will be substituted by their values. If there remain uninstantiated logic variables in the 'Smalltalk term', a run-time error will occur.

As a final remark on the 'rule-based interpreter' view of Figure 4.5, we draw the reader's attention to the fact that the 'Interpret' port of the 'Rule Interpreter' concept has two links attached to it. As both links are connected to a (port of a) different relation, this implies that the 'Rule Interpreter' concept participates in two different architectural relations via its 'Interpret' port. In Section 5.4, we will see that the semantics of an architectural view is the conjunction of all architectural relations in that view. In this particular case, the semantics of the two links attached to the 'Interpret' port of the 'Rule Interpreter' concept implies that there exists both an 'Updates' relation and a 'Uses data' relation between 'Rule Interpreter' and 'Working Memory'.

4.4 Application architecture

The previous architectural views described SOUL from the points of view of 'user interaction' and 'rule-based interpretation'. The architectural view discussed in this section focuses on the actual implementation structure of SOUL, such as how it is decomposed into classes and class hierarchies, and how these relate.

4.4.1 The SOUL class hierarchies

The complete syntax of the SOUL language is presented in BNF format in Appendix A. We will not go into the details of the language here. It suffices to say that the language is very similar to Prolog, modulo some minor syntactic differences and a special language construct for manipulating Smalltalk expressions.

The class decomposition of the SOUL implementation closely resembles the abstract syntax tree of the SOUL language. There are two main inheritance hierarchies in the SOUL implementation: SOUL clauses and SOUL terms. The 'clause' hierarchy contains classes representing SOUL facts, rules and queries. The full 'clause' hierarchy is shown below, where the indentation of the class names indicates their nesting in the inheritance hierarchy. For example, the class `SOULQuery` inherits from the class `SOULBasicClause`, which in turn inherits from `SOULClause`, but `SOULQuery` itself has no subclasses.

```
SOULClause
    SOULBasicClause
        SOULClauses
        SOULQuery
        SOULRule
            SOULCachedRule
            SOULFact
```

These clauses are typically built up from terms, which are defined by the 'term' hierarchy below.

```
SOULAbstractTerm
    SOULGeneratePredicate
    SOULTerm
        SOULNamedTerm
            SOULCompoundTerm
                SOULFixedNameCompoundTerm
                    SOULList
                    SOULPartialRepresentationList
            SOULVariableTerm
                SOULUnderscoreVariableTerm
        SOULSmalltalkConstantTerm
            SOULAdvancedSmalltalkTerm
                SOULSmalltalkMetaPredicate
            SOULCachedSmalltalkTerm
        SOULTerms
            SOULAndTerms
            SOULOrTerms
    SOULTrueTerm
```

4.4.2 The application architecture view

Figure 4.6 gives an overview of the entire 'application architecture' view of the SOUL system. We admit that the diagram is a bit dense. This is mainly due to the fact that we deliberately chose to represent all architectural relations explicitly (as rounded rectangles). The diagram could be simplified a lot by drawing all *Is Kinds Of*, *Has Part* and other relations, as special arrows that

connect the participating concepts. (For example, by using a UML-like notation.) However, as the main focus of this dissertation is on the underlying architectural formalism, we explicitly wanted to show all architectural entities on the picture. Nevertheless, we do think that an industrial-strength tool which incorporates our conformance checking formalism should provide support for simplified and customized graphical notations (see 8.3.4).

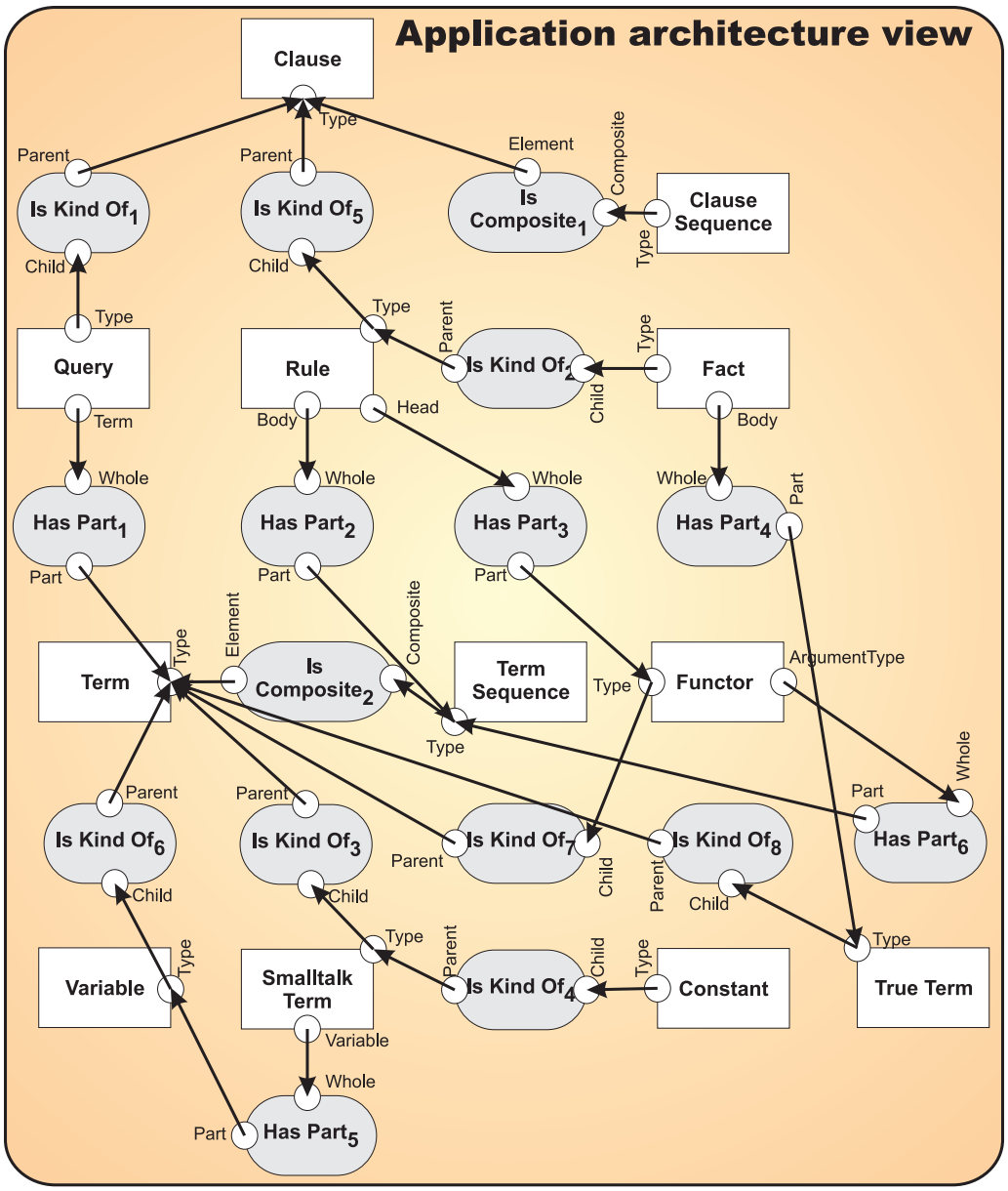

Figure 4.6: SOUL 'application architecture' view.

The only possible first-class expression in SOUL is a **Clause**, of which **Fact**, **Rule** and **Query** are special *kinds*. This is expressed by the *Is Kind Of* architectural relations. In fact, a **Fact** is a special kind of **Rule**, namely a rule with a 'true' body.

To allow SOUL applications to declare multiple facts and rules simultaneously, or a sequence of queries that should be executed one after the other, a **Clause** can also be a **Clause Sequence**, which in turn *contains* a number of **Clause**s. This relationship between **Clause** and **Clause**

Sequence is described by the *Is Composite* architectural relation. It expresses the fact that the composite element is both a *kind of* some other element type and that it is a *container* of elements of that type.

A **Term** is the most basic building block out of which the basic clauses **Fact**, **Rule** and **Query** are built. Terms can only be *part of* those clauses, and cannot occur as first-class entities in the SOUL language themselves. The 'term' class hierarchy describes the different kinds of terms that can be distinguished. Typical actions that all terms have in common are comparison, unification and interpretation.

In our 'application architecture', we only mention those terms that are considered being important. On the one hand, there is a need for some kind of **Term Sequence** that represents the body of a rule. Similar to a **Clause Sequence**, a **Term Sequence** is *composed* of other terms (in the sense of the composite pattern: it *contains* terms, and is itself a *kind of* term as well). This is necessary because the body of a rule is typically composed of many different terms. On the other hand, the head of a rule consists of a single **Functor** term, which is an atomic expression of the form $f(a_1, \ldots, a_n)$. In other words, functors have a name f, an arity n, which is a number, and a list of n arguments $a_1, \ldots, a_n$ which can either be instantiated or not. Thus, a **Rule** has two *parts*: a body, which is a **Term Sequence**, and a head, which is a **Functor**. Since a **Functor** has a list of arguments, which are all terms, it also has a **Term Sequence** as its part.

As already mentioned, a **Fact** can be considered as a special *kind of* **Rule** with a body that always succeeds. In order to be able to represent this, we need a **True Term** which corresponds to the Boolean value true. So although a **Fact** also has a head and a body as *parts*, its body can only be a **True Term**.

The most primitive kind of terms in SOUL are logic **Variable** and **Constant** terms. Besides variables and constants, SOUL also contains a notion of **Smalltalk Term**. SOUL rules and facts can use **Smalltalk Term**s to explicitly perform Smalltalk code during the logic interpretation process. **Smalltalk Term**s can have as *parts* **Variable** terms that are filled in at interpretation-time, to allow for a better symbiosis between SOUL and Smalltalk.

4.5 Summary

In this Chapter, we presented the case that is used throughout this dissertation. We described the architecture of the SOUL system from three different points of view. The 'user interaction' view focused on the interaction of a user with the SOUL system. The 'rule-based interpreter' focused on the interpretation of logic rules. The 'application architecture' was more implementation oriented and focused on the actual structure of the SOUL implementation.

For each of these architectural views, we used the same architectural notation. In the next chapter, this notation will be defined more formally. Moreover, we will explain how to map architectural views that use this notation to the implementation, so that conformance of the implementation to these architectural view can be checked. Using such an 'architectural mapping', we actually check conformance of the SOUL implementation to each of the three described views. In Chapter 7, we revisit these architectural views and show in detail what their architectural mappings look like.

Chapter 5

The Architectural Formalism

We explain both our architecture language and architectural conformance checking algorithm. An architectural model described in the architecture language declares the conceptual architecture of some software implementation, as well as the mapping of this declared architecture to the implementation artifacts and their dependencies. The architecture language consists of an architecture description language and an architectural mapping language. An architecture is described in the architecture description language. An architectural mapping is declared in the architectural mapping language. The conformance checking algorithm combines the architecture descriptions with the declared mappings to verify architectural conformance of some implementation.

5.1 Overview of the architecture language

The purpose of our *architecture language* is twofold. On the one hand, it describes what the architecture looks like, and on the other hand, it describes how the different architectural entities are mapped to the implementation. Obviously, how an architecture is mapped to an implementation depends strongly on the implementation under consideration, as well as on the chosen implementation language. Nevertheless, the architecture language itself is essentially independent of the chosen implementation, and to a certain extent also of the implementation language. While explaining the architecture language, we will clearly point out those parts that are implementation-language specific. In Chapter 7, we give a concrete example of how the architecture language can be used to describe and check a conformance mapping between a specific architecture and implementation.

Because the architecture language is used to describe an architecture as well as its mapping to the implementation, we split the language in two sub-languages: the *architecture description language* (ADL) and the *architecture mapping language* (AML). An architectural mapping defined in the AML again consists of several parts: an architectural instantiation, expressed in the *architectural instantiation language*, and an architectural abstraction, expressed in the *architectural abstraction language*. Furthermore, to define an architectural abstraction, an architect can use a library of predefined logic predicates. Following R. Wuyts [86], we call this library the *declarative framework*[1]

Describing the different submodels of an architectural model (i.e., its architecture description, architectural instantiation and architectural abstraction) separately enables a higher degree of reusability. For example, to describe another software system with a similar architecture, we might want to reuse only the description of the architecture (or parts of it). Or we might want to reuse some architectural abstraction to map it to other architectural entities, or to define other architectural abstractions.[2]

[1] This terminology is taken from [54, 86]. Recall from 4.1 that the logic language SOUL provides a layered library of rules at several levels of abstraction, ranging from a primitive Smalltalk-specific layer to higher-level layers for reasoning about design patterns and architectural constraints. Wuyts calls this library a 'declarative framework'.

[2] For example, in Subsection 7.2.1 we will show how the same virtual classification can be mapped to two different

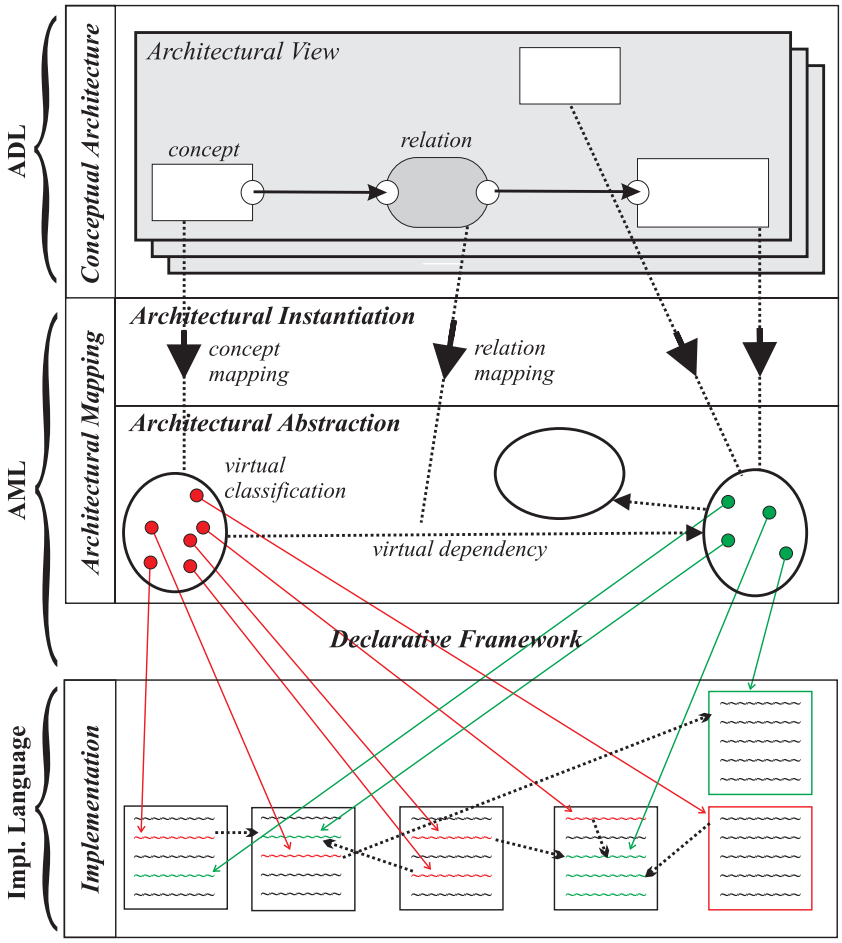

Figure 5.1: Schematic overview of the architecture language.

Figure 5.1 presents a schematic overview of how an architecture and its mapping to the implementation are represented in our architecture language:

Architecture Description Language The ADL can be used to define the *conceptual architecture* of a software system. The conceptual architecture describes a software system from multiple high-level architectural point(s) of view, abstracting away the implementation details of the system. Each architectural view focuses on a different aspect of the structure of the software system.

The ADL defines only the syntax of a conceptual architecture. It allows us to name the architectural entities in the different architectural views and to describe how they are interconnected. The semantics of these entities is not defined in the ADL, but is described in the AML.

Architectural Mapping Language The AML allows us to codify the mapping to the implementation for each of the architectural views described in the ADL, thus defining the meaning of the different architectural entities in each of these views. Every architectural entity is defined in terms of implementation artifacts and their dependencies.

architectural concepts belonging to different architectural views. Another example (see Subsection 7.1.3) will show how a virtual classification can be used both as an architectural abstraction for defining an architectural concept and as an auxiliary building block for defining another virtual classification.

An *architectural mapping* expressed in the AML consists of an *architectural instantiation* and an *architectural abstraction*. An architectural instantiation merely associates architectural entities with intermediary abstractions defined in the architectural abstraction language. These intermediary abstractions define the actual mapping to the implementation. But instead of having to define these abstractions directly in terms of implementation artifacts and their dependencies, the AML provides a *declarative framework* of predefined auxiliary predicates that can be used to define an architectural abstraction.

Architectural instantiation language In this language, we can map architectural entities defined in the ADL to the intermediary abstractions defined in the architectural abstraction language.

Architectural abstraction language This language provides intuitive high-level abstractions of sets of implementation artifacts and their dependencies that can straightforwardly be mapped to the different kinds of architectural entities of the ADL.

Declarative framework (DFW) This is a layered library of predefined logic predicates ranging from very high-level predicates for defining architectural mappings to very primitive predicates that reason about artifacts in the implementation language. These predefined predicates are defined in terms of more primitive predicates, which are in turn defined in terms of even more primitive predicates. Table 5.1 summarizes the different layers of the declarative framework. Each of these layers, together with the predicates they provide, will be discussed in more detail in Subsections 5.3.3 to 5.3.6.

Architectural layer	
	Architectural mapping predicates
	Predefined predicates for defining architectural abstractions
Implementation layer	
	Design patterns layer
	Predicates for codifying design patterns and styles
	Coding conventions layer
	Predicates for codifying coding conventions and styles
	Base layer
	Predicates for reasoning about implementation artifacts and their dependencies
	Representational layer
	Predicates for retrieving artifacts from the implementation repository
Logic meta-programming layer	
	Repository access
	Language-independent predicates for accessing the implementation repository
	Logic layer
	Built-in predicates of the logic language

Table 5.1: Layers of the declarative framework.

The ADL is entirely independent of the particular software system under consideration[3] and even of the implementation language that was used. The same holds for the architectural instantiation language and architectural abstraction language. Only the declarative framework, which contains a set of predefined predicates to reason about the implementation, relies on the chosen implementation language. Some layers of the framework are more dependent on the implementation language than others, however. And even the declarative framework is essentially independent of the particular implementation under consideration. Of course, the syntax and implementation of the predicates provided by the declarative framework do depend on the chosen LMP language. But a similar set of predicates could be defined in any other declarative language.

[3] In other words, it is an 'implementation independent' ADL.

5.2 The architecture description language (ADL)

In this section, we introduce the ADL in which the conceptual architecture of a software system can be described. The conceptual architecture describes the architecture of a software system at an abstract conceptual level, ignoring the details of implementation, algorithm, and data representation of that system. Our ADL is essentially the same as the one used by Shaw and Garlan [74]. Their approach to architectural representation is based on seven kinds of entities: components, connectors, configurations, ports, roles, representations and bindings. The last two are needed to allow nesting of sub-architectures inside components or connectors. Essentially the same constructs (though sometimes with a different name) form the basis of our ADL: *concepts*, *relations*, *links*, *ports*, *roles*, *sub-architectures* and *bindings*, respectively. Additionally, we also have *architectural views*.

We preferred to use an existing ADL, rather than defining yet another one. We chose Shaw and Garlan's notation, because it is simple and general. We agree with J. Kramer and J. Magee [37] that a language for describing architectural structures should be simple and concise, and explicitly designed for this purpose. This is the case for Shaw and Garlan's ADL. The different language constructs needed for defining a conceptual architecture are explained informally below.

Conceptual architecture Because we want to allow multiple architectural views, each focusing on a different aspect of the structure of some software system, a global name space in which all these architectural views reside is needed. We will call this set of architectural views the *conceptual architecture* of the system. For example, in the case of the SOUL system, the conceptual architecture consists of a 'user interaction' architectural view, a 'rule-based interpreter' architectural view and an 'application architecture' view.

Architectural views Every *architectural view* has a name (which is unique within its conceptual architecture) and groups a set of architectural concepts and relations with the links that glue them together. In Chapter 4, we encountered several examples of architectural views.

Architectural concepts and relations We distinguish two kinds of architectural elements: *concepts* and *relations*. Whereas the architectural concepts represent the concepts of interest in a particular architectural view, the architectural relations describe the important relationships between these concepts. Examples of architectural concepts in the 'user interaction' architectural view are 'Input Window', 'User Application' and 'Query Result'. Examples of architectural relations are 'Asks', 'Activates' and 'Is Created By'.

Ports and roles Every architectural element contains a set of gates representing the external interface of that element. For example, for an architectural concept representing an Input Window, a gate may represent the events that can be generated by this window. Every gate belongs to exactly one architectural element and has a name that is unique in that element. The gates of an architectural concept are called *ports* and represent the interaction points of that concept with its environment. The gates of an architectural relation are called *roles* and identify the participants of that relation.

Links Architectural elements are connected by means of *links* that map element gates to other element gates. The presence of a link between two gates expresses that there is some flow of information or control between the gates. Links always connect a concept port with a relation role and can only relate gates that belong to elements defined in the same architectural view.

Although links are essentially undirected, in our pictures we often do put arrows on the links. As explained in Subsection 4.1.3, these arrows have no semantics and are only meant to make the diagrams more readable.

Only one link can exist between two gates. It is allowed however, to have multiple links attached to the same port (resp. role), provided that all linked roles (resp. ports) are different. When multiple links are attached to a single relation role, we interpret this as

a disjunction: the relation should hold for at least one of the concepts linked to this role. Two examples of this in the 'user interaction' architectural view are the 'Action' role of the 'Activates' relation and the 'Creator' role of the 'Is Created By' relation (see Subsection 4.2.2). Recall from 4.2.2 that if we want a conjunctive interpretation rather than a disjunctive one, we explicitly need two separate relations, as the semantics of an architectural view is the conjunction of the semantics of all its relations.

Sub-architectures and bindings. In Subsection 6.4.5 we will explain how composite architectural concepts and relations can be defined in terms of sub-architectures and a notion of *bindings*.

Figure 4.1 (page 42) illustrates the graphical representation of some of the entities of a conceptual architecture. It depicts a subset of the 'user interaction' architectural view containing three concepts: **Input Window**, **User Application** and **Auxiliary Application**, and two relations: *Activates* and *Is Created By*. The **User Application** concept has two ports: Request and Type, through which it interacts, via the relations, with the Event port of the **Input Window** and the Type port of the **Auxiliary Application** concept, respectively. The *Activates* and *Is Created By* relations each have two roles identifying the two actors (Trigger and Action, and Created and Creator, respectively) playing a role in the relation.

Some important remarks should be made about this example. First of all, it illustrates that the same role can be connected to more than one port. Secondly, the example seems to suggest that architectural relations always represent "actions". This is not necessarily the case. For example, in the 'application architecture' view of SOUL (see 4.4.2), the architectural relations represent structural relationships such as an "is a" or "part of" relationship. Finally, the example seems to indicate that ports and or roles may have some kind of meaning attached to them. However, in the conceptual architecture, a port or a role only have a name. A meaning is attached to them by defining an architectural mapping in the AML.

5.3 The architectural mapping language (AML)

In order to allow conformance checking of the implementation of a software system to the architectural views declared in a conceptual architecture, a mapping between the declared architectural views and the implementation needs to be established. This mapping is defined in the AML. As explained in Section 5.1, the AML is split into three parts. An architectural instantiation is declared in the 'architectural instantiation language' and maps architectural entities in the conceptual architecture to architectural abstractions. These architectural abstractions are described in the 'architectural abstraction language' and define the actual mapping to the implementation. Rather than defining the architectural abstractions directly in terms of implementation artifacts and dependencies, they are defined in terms of the predefined predicates provided by the 'declarative framework'. The following subsections elaborate on the architectural abstraction language, the architectural instantiation language and the declarative framework. Table 5.2 summarizes the different constructs of each of these languages and shows how they relate to the constructs of the ADL.

	Architectural Mapping Language		
ADL	Arch. Inst. Lang.	Arch. Abstr. Lang.	DFW
architectural view			
concept	concept mapping	virtual classification	"*predefined*
relation	relation mapping	virtual dependency	*architectural*
port	port mapping	filter	*mapping*
role	role mapping	argument number	*predicates*"
link	link mapping	quantifier	

Table 5.2: Overview of the architectural mapping language.

5.3.1 The architectural abstraction language

The architectural abstraction language we propose in this dissertation is a refinement of the mapping language we proposed in an earlier paper [52]. As in that paper, we will associate architectural concepts with virtual classifications that compute a set of implementation artifacts that corresponds to those concepts. Architectural relations will be mapped to virtual dependencies expressing high-level implementation relationships that will be applied to the elements of the virtual classifications.

Architectural abstraction	Semantics in terms of implementation artifacts and dependencies
Virtual classification	Computed set of implementation artifacts, such as: classes, methods, variables, ...
Virtual dependency	High-level relationship among implementation artifacts
Filter	Function selecting a subset from a set of artifacts
Argument number	Number referring to one of the arguments of a logic predicate
Quantifier	Quantifier over a set of artifacts

Table 5.3: Constructs of the architectural abstraction language.

Below, we informally explain each of the different constructs of the architectural abstraction language. Table 5.3 summarizes them.

Virtual classifications A virtual classification groups a set of related implementation artifacts. It is virtual in the sense that the elements of the set are not explicitly enumerated. Instead, the set is declared intentionally, by means of some high-level logic predicate, so that it can be computed when needed. Such virtual classifications provide an interesting abstraction

in which terms to model architectural concepts. They have intuitive appeal and have a concise representation. They are also interesting from the perspective of evolution: even when the implementation changes, the virtual classifications may still compute the intended artifacts, thanks to their intentional declaration. Finally, they are particularly well-suited to model concepts of architectural views that 'cut across' the implementation structure. One implementation artifact may belong to multiple virtual classifications, and one virtual classification can contain many implementation artifacts that are spread throughout the code.

As an example, consider the **Query Interpreter** concept in the 'user interaction' architectural view of the SOUL system. The virtual classification associated with this concept computes all Smalltalk classes and methods that address the concern of 'interpreting queries'. Such a classification can be defined in many ways: by simply enumerating all these classes and methods; by making use of some naming conventions, for example that they all have a name which starts with the same prefix 'interpret'; by using explicit tagging information or structured documentation in the source (e.g., all methods in the Smalltalk method protocol named 'interpretation'); or by using some semantic inferencing, such as that they are all invoked or accessed by some method which is supposed to start the interpretation process.

Virtual dependencies Whereas virtual classifications model architectural concepts, virtual dependencies are the constructs for modeling architectural relations. Because architectural relations intuitively represent the interaction among architectural concepts, we will model them as high-level relationships over these concepts. More precisely, since architectural concepts are modeled as (computed) sets of implementation artifacts, we will specify only how the artifacts in the different sets should be related. Hence, a virtual dependency is some high-level design or implementation relationship among implementation artifacts. It is virtual because it may require some computation to derive it from the implementation.

Filters In the ADL, concept ports intuitively represent the external interface of a concept. Because concepts are modeled in terms of (computed) sets of implementation artifacts (i.e., virtual classifications), we model concept ports as filters over these sets. Such a filter 'deletes' all information from the set that is not relevant for a particular port. This corresponds to the intuition that a concept port is a kind of peep-hole through which (only) part of the internals of the concept can be seen.

For example, consider the **User Application** concept in the 'user interaction' architectural view. This concept classifies all implementation artifacts that model applications in the SOUL system. In particular, for each type of user application it contains a class implementing that application, as well as all methods of that class. The **User Application** concept has two ports: Request and Type. Intuitively, the Type port models all possible types of user applications and the Request port models all possible requests that can be sent to those applications. Because in the Smalltalk implementation of the SOUL system, the user applications are modeled as classes, we associate with the Type port a 'class filter' which filters only the classes from the classification. Similarly, because the methods of these classes represent the requests the applications can handle, we associate with the Request port a 'method filter' to filter only the methods from the classification.

Filtering serves two purposes. On the one hand it will improve the efficiency of the conformance checking algorithm, by reducing the number of elements of a virtual classification that need to be considered. Indeed, to avoid having to check all elements of a virtual classification, relations are not linked to a concept directly, but to one of its ports. On the other hand, filtering may be interesting for typing purposes. For example, after applying a 'class filter' only artifacts of type 'class' remain. This knowledge is useful when defining virtual dependencies.

Argument numbers Architectural relations are represented by logic predicates representing high-level dependencies among implementation artifacts. Relation roles correspond to the arguments of those predicates. We need to define precisely which role corresponds to which argument. To define this correspondence, we use the argument numbers.

For example, we will see in Subsection 7.1.5 that the virtual dependency associated with the *Is Created By* architectural relation in the 'user interaction' view, is defined by the logic predicate `isCreatedBy_C_C(Class1, Class2)`. In this predicate, the first argument represents the created class, and the second argument represents the creating class. Therefore, the 'Created' role on the *Is Created By* architectural relation corresponds to argument number 1, and the 'Creator' role corresponds to argument number 2. (Note that we did not mention these argument numbers on the figures, in order not to clutter them too much.)

Quantifiers Since a virtual dependency does not work directly on sets of artifacts, but on single artifacts, we need a way of generalizing it to a relationship over sets. To this extent, we need some more information such as: do we need to consider all artifacts in the set, or is it sufficient to check the relationship for only one artifact? This information is specified by the *quantifiers* to which the links are mapped. They state how a virtual dependency should be 'applied' over the elements of a virtual classification. Typical quantifiers are set quantifiers such as ∀ and ∃. Figure 5.2 shows a refined version of Figure 4.4 where every link has been annotated explicitly with a quantifier.

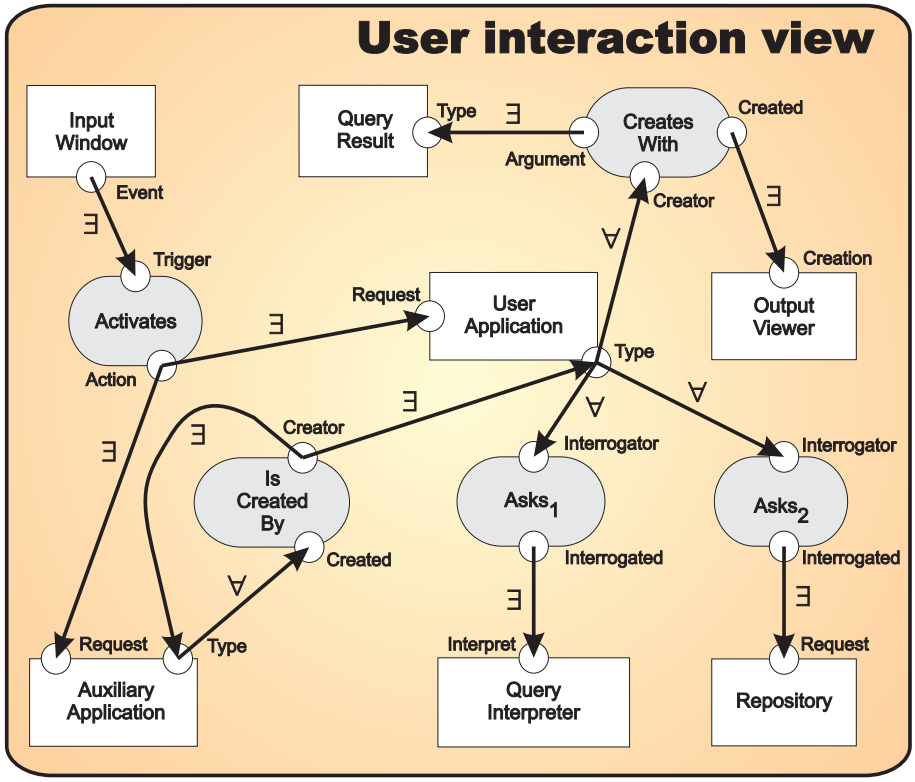

Figure 5.2: The 'user interaction' architectural view with quantifiers.

For example, consider the $Asks_1$ architectural relation between the architectural concepts **User Application** and **Query Interpreter** in the 'user interaction' architectural view. This architectural relation has two links, one for the role 'Interrogator' and one for the role 'Interrogated', with associated quantifiers ∀ and ∃ (respectively). These quantifiers should be interpreted as follows: "*every* interrogator element (i.e., every type of user application)

asks for information to *at least one* interrogated element (i.e., some query interpretation request)".

One could argue that the quantifiers should not be specified by the architectural mapping but by the architecture description, because they contain important information on how to interpret an architecture. Furthermore, this information is independent of the implementation under consideration. For now, however, we declare the quantifiers only as part of the architectural mapping. In Subsection 5.4.6 we will explain what changes are needed to the formalism if we would want to include quantifiers in the ADL.

This concludes our definition of the various constructs of the architectural abstraction language. Subsection 6.2.2 will explain how each of these constructs are represented in our LMP medium. We stress that none of these constructs make special assumptions about the kinds of implementation artifacts that are considered. Therefore, the architectural abstraction language is independent of the chosen implementation and implementation language. The same will be the case for the architectural instantiation language.

5.3.2 The architectural instantiation language

We have explained the ADL, which allows us to define the architectural concepts, relations, ports, roles and links in the different architectural views, and the architectural abstraction language, in which we can define virtual classifications, virtual dependencies, filters and quantifiers. Now we explain the architectural instantiation language, which can be used to define architectural instantiations. An architectural instantiation associates architectural concepts with virtual classifications, architectural relations with virtual dependencies, ports with filters, roles with argument numbers and links with quantifiers. To define these associations, the architectural instantiation language provides the following language constructs:

Concept mapping Concept mappings associate architectural concepts with virtual classifications, thus defining the set of implementation artifacts that corresponds to each of those concepts.

Port mapping As architectural concepts are mapped to sets of implementation artifacts, we map concept ports to filters that select a relevant subset from the set associated with the concept.

Relation mapping An architectural relation is mapped to a virtual dependency which expresses the relation among the artifacts classified according to the concepts linked to the architectural relation.

Role mapping The roles of an architectural relation are mapped to argument numbers of the virtual dependency that is associated with the architectural relation.

Link mapping Link mappings are used to associate a quantifier with each link.

The mappings that can be defined in the architectural instantiation language are very simple. Every entity in a conceptual architecture is mapped to an architectural abstraction. Because of this trivial mapping, one might argue that the architectural instantiation language is not really necessary. Instead, we could adopt an implicit mapping, by using the same names for concepts in the conceptual architecture as for the virtual classifications in the architectural abstraction, and likewise for the relations and ports. However, this alternative approach has a number of shortcomings. For example, what if we need two different architectural relations that correspond to the same virtual dependency? We cannot use the same name, because we need to distinguish them (they can be linked to different concepts). And what if we have a concept with different ports that are associated with the same filter? It may not be opportune to combine the two ports to one single port, just because they are instantiated with the same filter. The fact that they both correspond to the same filter may be an accidental feature of the instantiation for a particular

implementation, and could be different when the architecture is instantiated for another system. Using an explicit architectural instantiation avoids such problems by making explicit the mapping between a conceptual architecture and the architectural abstractions.

To conclude, we point out that the architectural instantiation language is completely independent of the chosen implementation and implementation language. It just maps ADL constructs to constructs of the architectural abstraction language. No assumptions whatsoever are made of the underlying implementation (language).

5.3.3 The declarative framework (DFW)

Architectural abstractions define the actual mapping of architectural entities to implementation artifacts and their dependencies. Because we use a LMP approach, when defining a particular architectural abstraction (e.g., a particular virtual classification) we can take advantage of the full expressive power of a logic meta-programming language. Technically speaking, it is almost obvious that, given some meta-programming language that is sufficiently expressive, it is possible to define any architectural mapping. In practice, however, defining a particular architectural mapping is not trivial, because many architectural constraints are implicit in the source code and because there is often no one-to-one mapping of architectural abstractions to implementation entities.

Therefore, to aid us in defining architectural abstractions, in addition to having an expressive LMP language, we would like to have some library of predefined predicates which capture the most common mapping schemes for the different kinds of architectural abstractions. In other words, when defining most architectural mappings, it should be sufficient to just select the appropriate logic predicate(s) from the library and fill in the required parameters. We call this library the declarative framework (DFW). This library has an explicit layered structure where the predicates in the higher layers are defined in terms of those in the lower layers. The terminology 'declarative framework' stems from [54, 86].

For some architectural mappings, the framework may not (yet) provide predefined predicates that codify these mappings. In these cases, we can rely on the expressive power of the LMP language to define our own predicates (which can be added to the DFW so that they can be reused later on to define similar architectural mappings). When defining such predicates, the framework may offer some help, because of its layered structure. The predicates it provides are distributed over several layers of abstraction. So even if we do not find a particular predicate we need at a certain level of abstraction, the abstraction level below may very well provide the predicates we need to define the required predicate. In other words, we do not necessarily have to descend all the way down to the implementation layer to define our architectural mapping.

Below, we briefly introduce this layered declarative framework. We enumerate the different layers and explain the kind of predicates each such layer contains. We explain the most primitive layers first, and then proceed with the more abstract ones. The layers were subdivided in three main groups: a logic meta-programming layer, an implementation layer and an architectural layer. Figure 5.3 on page 95 provides an overview of some of the layers of the DFW and illustrates how predicates in the higher-level layers make use of predicates defined in the lower-level ones.

Logic meta-programming layer A first technical layer provides some primitive predicates of the LMP language, that is, some primitives of the logic language as well as some primitive predicates for accessing the implementation repository.

- **Logic layer** The bottom layer merely contains primitive logic predicates that can be found in many logic languages, such as `not`, `findall`, `read`, etc.
- **Repository access** Next, we have a layer containing primitives for accessing the repository containing the implementation artifacts. These predicates can be used to access information in the repository, but are independent of the actual kind of information that is stored in that repository. For example, if the repository is a database, a primitive predicate `addRecord` is provided to add a record to some table in the database.

Implementation layer The implementation layer defines the most frequently-used predicates for reasoning about artifacts in some implementation language. Of course, this layer highly depends on the chosen implementation language. In our experiments, we used Smalltalk.

- **Representational layer** First of all, the representational layer is a Smalltalk-specific layer that contains predicates for retrieving Smalltalk artifacts from the repository. For example, a predicate `class` can be used to retrieve a Smalltalk class from the repository (or to check whether a certain class can be found in the repository).
- **Base layer** The base layer adds a whole range of predicates to the representational layer, to facilitate reasoning about the implementation. Typing predicates, such as `instVarTypes`, that infer the types of instance variables and other Smalltalk expressions are an example of typical functionality offered by the base layer.
- **Coding conventions** Built on the base layer, we have several other layers containing even higher-level predicates. One such layer codifies the typical coding conventions and styles that Smalltalk programmers use. It also contains some more generic predicates such as the predicate `findMethod` which can be used to find all methods that match a certain pattern.
- **Design patterns** Another, even more abstract level, codifies rules for capturing design patterns. For example, the predicate `compositePattern` checks for occurrences of the Composite design pattern.

Architectural layer Finally, the architectural layer provides some predicates that capture the most common ways of mapping architectural abstractions to implementation artifacts and their dependencies.

- **Architectural mapping predicates** This layer defines some auxiliary and template predicates for defining architectural abstractions. E.g., the `findMethodsFromClasses` predicate can be used to compute the methods that should belong to some virtual classification, based on the classes that already belong to that classification.

From the above descriptions, it should already be clear that the implementation layer highly depends on the fact that the chosen implementation language is Smalltalk. Nevertheless, even in this layer, we try to reduce the Smalltalk-dependence to a minimum. We take advantage of the layered structure of the framework to restrict the Smalltalk-specific predicates to the lowest layers as much as possible.

It is also important to realize that the DFW is not really specific to our architecture language. It is a general library of logic predicates that can be used for any kind of declarative reasoning about Smalltalk source code. In fact, the same DFW is currently being used in other contexts by other researchers at our lab. Only the architectural layer of the DFW contains some predicates that are specific for our experiments on architectural conformance checking.

In Subsections 5.3.4 to 5.3.6, we discuss each of the layers of the DFW in more detail, again starting with the least abstract ones. For each layer we also mention to which extent it depends on the fact that the chosen implementation language is Smalltalk.

5.3.4 The logic meta-programming layer of the DFW

The logic meta-programming layer consists of two sublayers: the logic layer and the repository-access layer. All predicates in the logic meta-programming layer are independent of the chosen implementation language.

Logic layer

The most primitive layer is the logic layer which contains the primitive logic predicates that are provided by most logic languages. It includes predicates defining arithmetic functions (`is`, `<`,

`>`, ...), program control (`not`, `call`, `findall`, `forall`, ...), data handling (list handling, string handling, term type checking, ...), input and output (`read`, `write`, ...), etc. We will not discuss the details of this layer here; instead we refer to a Prolog manual [85].

Because the predicates in this layer are only primitive logic predicates that do not support reasoning about the implementation artifacts in some base language, they are clearly independent of the chosen base language.

Repository-access layer

The repository-access layer defines some very specific predicates for accessing an implementation repository. For example, if the implementation repository would be an ODBC-compliant database, the repository-access layer would contain some general predicates that use ODBC primitives to translate the information in the database to logic terms. The repository-access layer is defined on top of the logic layer, because it uses some of the primitives provided by that layer. Because of the very technical nature of the repository-access layer, we do not explain it in detail here. More technical details on how the logic meta language can access the implementation repository (i.e., which languages, environments, interfaces, repositories and tools were used, and how they were combined), are given in Section 6.1 and Subsection 8.4.3.

This layer is also independent of the chosen base language. Although the repository-access layer provides predicates for accessing the information stored in a repository, these predicates are still independent of the actual kind of information that is stored in that repository.

5.3.5 The implementation layer of the DFW

The implementation layer provides a whole range of predicates for reasoning about implementations in the Smalltalk language. Obviously, many of these predicates are Smalltalk-specific, although some may be valid for other object-oriented languages as well. However, they are not dependent on the particular implementation that is considered. They can be used for reasoning about any implementation in Smalltalk.

The implementation layer consists of three sublayers: the representational layer, the base layer, the coding conventions layer and the design patterns layer.

Representational layer

In order to check conformance of an implementation to a described architecture, our DFW should be able to reason about the implementation artifacts and structures in the implementation repository. This is the responsibility of the 'representational layer'. It defines the meta-level interface between the LMP language and the underlying base language. In our case, the base language is Smalltalk. Therefore, this layer contains a set of predicates for retrieving Smalltalk artifacts (such as classes, methods and instance variables) and their structural relationships (such as inheritance) from the implementation repository. All layers that are defined on top of the representational layer use this layer to reason about the implementation. The predicates of the representational layer make use of the underlying repository-access layer to access the implementation repository.

Table 5.4 lists some of the predicates provided by the representational layer. We distinguish three different kinds of predicates:

1. predicates that retrieve a Smalltalk artifact from the repository;
2. predicates that select the name of some Smalltalk artifact;
3. predicates that reason about structural relationships among artifacts.

From the table, we see that all predicates are specifically targeted towards reasoning about an object-oriented base language. Some predicates, like `category` and `protocol` are specific to

Predicate name and arguments	**Meaning of the predicate**
Retrieving implementation artifacts	
`class(Cl)`	`Cl` is a Smalltalk class or meta class
`baseClass(Cl)`	`Cl` is a Smalltalk class (but not a meta class)
`metaClass(MC)`	`MC` is a Smalltalk meta class
`method(Me)`	`Me` is a Smalltalk method
`instVar(IV)`	`IV` is a Smalltalk instance variable
`temporaryVar(TV)`	`TV` is a temporary variable of a method
`argumentVar(AV)`	`AV` is a method argument
`category(Ca)`	`Ca` is a Smalltalk class category
`protocol(Pr)`	`Pr` is a Smalltalk method protocol
...	...
Selecting implementation artifact names	
`className(Cl,CN)`	Class `Cl` has name `CN`
`methodName(Me,MN)`	Method `Me` has name `MN`
`instVarName(IV,IN)`	Instance variable `IV` has name `IN`
`argumentVarName(AV,AN)`	Method argument `AV` has name `AN`
`temporaryVarName(TV,TN)`	Temporary variable `TV` has name `TN`
`categoryName(Ca,CN)`	Class category `Ca` has name `CN`
`protocolName(Pr,PN)`	Method protocol `Pr` has name `PN`
...	...
Structuring of implementation artifacts	
`classImplementsMethod(Cl,Me)`	Class `Cl` implements method `Me`
`classImplementsMethodNamed(Cl,MN,Me)`	Class `Cl` implements method `Me` with name `MN`
`inheritance(C1,C2)`	Class `C2` inherits from class `C1`
`metaClass(Cl,MC)`	Class `Cl` has meta class `MC`
`methodParseTree(C,M,A,T,S)`	*Computes a method parse tree*
`instVar(Cl,IV)`	Class `Cl` has instance variable `IV`
`methodArgument(Me,AV)`	Method `Me` has argument `AV`
`methodTemporary(Me,TV)`	Method `Me` has temporary variable `TV`
`classInCategory(Ca,Cl)`	Class `Cl` belongs to class category `Ca`
`methodInProtocol(Cl,Pr,Me)`	Method `Me` in class `Cl` belongs to method protocol `Pr`
...	...

Table 5.4: Some predicates provided by the representational layer.

Smalltalk, whereas others, like `class`, `method` and `inheritance` are relevant for other object-oriented languages as well.

Because of the multi-way reasoning capabilities of our logic meta language, all these predicates can be used in multiple ways, depending on which of their arguments are left uninstantiated. For example, a predicate `classImplementsMethod(Cl,Me)` can be used in four different ways: to check whether some specified class implements some specified method, to compute all classes that implement some specified method, to compute all methods that are implemented by some class, or to compute all class-method pairs such that the class implements the method.

A particularly important predicate of the representational layer is:

```
methodParseTree(ClassName, MethodName, ArgumentList, TemporariesList, StatementList)
```

This predicate takes a `ClassName` and `MethodName` as input (or they can be left uninstantiated, in which case the appropriate values are generated for them) and returns the different parts of the method parse tree for the method named `MethodName` and belonging to a class named `ClassName`. Both the list of method arguments (`ArgumentList`) and the list of temporary variables of that

method (`TemporariesList`) are returned, as well as the parse tree (`StatementList`) of the method in a structured format.
For example, consider the following simple Smalltalk method which belongs to the class `SOULTerms`:

```
at:  anInteger
    ^self terms at:  anInteger
```

For this method, the query `methodParseTree('SOULTerms','at:',Args,Tmps,Tree)` would return the following result:

```
Args = [anInteger]
Tmps = []
Tree = [return(send(send(variable(self),terms,[]),'at:',[variable(anInteger)]))]
```

Based on the `methodParseTree` predicate, we can define many useful predicates that reason about the implementation structure. In fact, *`methodParseTree`* is a generic predicate that is only supposed to be used as an auxiliary predicate to define other predicates. (That is why we printed it in italics in Table 5.4.)

Base layer

The representational layer is a very primitive layer defining only the most primitive logic predicates for reasoning about the underlying Smalltalk base language. On top of this layer, a more elaborate 'base layer' is defined, which extends these predicates with some extra predicates that are often used for reasoning about the implementation artifacts and their dependencies. All predicates in this base layer are defined directly in terms of predicates of the representational layer. They do not access the implementation repository or the repository-access layer directly. We distinguish three groups of predicates in the base layer:

1. predicates that are defined in terms of a method parse-tree traversal;
2. predicates that infer the type of certain Smalltalk expressions;
3. predicates that implement complex structural relationships.

Each of these groups is discussed below and summarized in Table 5.5. For the actual implementation of (some of) these predicates, we defer to Subsection 6.2.4 and Chapter 7.

Method parse-tree traversing. To reason about the structure of Smalltalk methods, the representational layer provides a predicate `methodParseTree`. Based on this predicate, many useful base-layer predicates can be defined which reason about methods. For example, a predicate `isSentTo` which checks for an invocation relationship can be defined by examining a method parse tree in search for method sends that occur in the body of the method. Similar predicates `assignStatement` and `returnStatement` can be defined to check for a variable assignment or a return statement, respectively. Because all these predicates exhibit many similarities, we decided to factor out their commonalities in a very general method parse-tree traversal predicate

```
traverseMethodParseTree(ClassName,MethodName,Environment,Found,Process)
```

This predicate traverses the parse tree of a method named `MethodName` in some class named `ClassName`, looking for some information to be stored in (or checked against) the variables in the passed `Environment`. This predicate is a second-order logic predicate as it takes two first-order predicates `Found` and `Process` as argument. These first-order predicates define the kind of information we are looking for in the parse tree (`Found`) as well as how this information should be processed (`Process`) to extract the required pieces of information to be accumulated. Using this predicate `traverseMethodParseTree` we can easily define (amongst others) the following predicates:

Predicate name and arguments	**Meaning of the predicate**
Method parse-tree traversing	
traverseMethodParseTree(C,M,E,F,P)	*Traverse a method parse tree*
isSentTo(CN,MN,Rcvr,Msg,Args)	Find message sends in a method
assignStatement(M,Var,Val)	Find variable assignments in a method
returnStatement(CN,MN,E)	Find return statements in a method
...	...
Type inferencing	
mayHaveType_E_M_C(E,M,C)	Infer type C of expression E in method M
mayHaveType_V_M_C(V,M,C)	Infer type C of variable V in method M
returnType(M,C)	Infer return type C of method M
instVarTypes(C,IV,TL)	Infer the valid types TL for an instance variable IV of class C
classVarTypes(Cl,CV,TL)	Infer the valid types TL for a class variable CV of class Cl
temporaryTypes(M,V,TL)	Infer the valid types TL for a temporary variable V of method M
argumentTypes(M,A,TL)	Infer the valid types TL for an argument A of method M
...	...
Structural relationships	
closure(Relation,Term1,Term2)	*Compute transitive closure of a binary relation*
hierarchy(Super,Sub)	Class Sub belongs to the class hierarchy of Super
instVarFlattened(Cl,IV)	Class Cl or a superclass has instance variable IV
understands(Cl,Msg)	Class Cl understands message Msg
...	...

Table 5.5: Some predicates provided by the base layer.

- isSentTo(ClassName,MethodName,Receiver,Message,Arguments) checks whether some method named MethodName in a class named ClassName sends some Message with some list of Arguments to some Receiver class.
- assignStatement(Method,Variable,Value) checks whether some Method assigns some Value to some Variable.
- returnStatement(ClassName,MethodName,Expression) checks whether a method named MethodName in a class named ClassName returns some Expression.

Just like the generic predicate *methodParseTree* of the representational layer, the base-layer predicate *traverseMethodParseTree* was printed in italics in Table 5.5: it is a generic predicate that is only supposed to be used as an auxiliary predicate to define other predicates.

Type inferencing. Smalltalk is a dynamically typed language, meaning that type information is not explicitly declared in the source code and that types are only checked at run-time (i.e., 'message not understood' errors are generated when invalid messages are sent to a certain expression). Nevertheless, type information is often useful when reasoning about the implementation. Therefore, the base layer includes a set of typing predicates that 'guess' the types of variables and other Smalltalk expressions. Since Smalltalk is a pure object-oriented language, we consider classes as types, and say that the type of an object is the class that implements it.

- Our ability to reason about type information is limited because of the absence of explicit type declarations in Smalltalk, and because we reason only about the static structure of the

implementation and do not take run-time information into account. But even within these constraints, we managed to implement a predicate `mayHaveType_E_M_C(E,M,C)` to infer the type `C` of some expression `E` that occurs in the body of some method `M`. The predicate was implemented as follows: to infer the type of an expression, we take a look at all messages that are sent to that expression (in the context where it occurs). In order for a class to be a valid type for that expression, it should understand at least all these messages (if not, a 'message not understood' error may occur at run-time). Every class that does understand all these expressions is a possible type.

Unfortunately, sometimes there are multiple candidate classes[4] for this type, and it is impossible to decide which one is the correct choice, without using dynamic information or doing extensive (and costly) data-flow analysis or type inferencing. Hence, the described approach only provides an approximate answer, but it is the best we can do under the given circumstances. (The more messages are sent to an expression, the more precise the answer will be.)

- The predicate `mayHaveType_V_M_C(V,M,C)` is a more specific version of `mayHaveType_E_M_C` where the expression is a variable `V` (instance variable, class variable, temporary variable, method argument, ...) occurring in the body of method `M`.
- A predicate `returnType(M,C)` which checks whether a method `M` returns an object of type `C` can be defined straightforwardly in terms of `mayHaveType_E_M_C` and `returnStatement`.
- Using a similar technique as for `mayHaveType_E_M_C`, the `instVarTypes(C,IV,Types)` predicate computes a list of possible `Types` for an instance variable `IV` of some class `C`.
- The predicates `classVarTypes`, `temporaryTypes` and `argumentTypes` for inferring the types of class variables, temporary variables and method arguments are defined similarly.

Structural relationships. The representational layer contains some very primitive predicates for reasoning about structural relationships among Smalltalk implementation artifacts. For example, it defines a predicate `classImplementsMethodNamed` for checking whether a class implements a method with a certain name, a predicate `instVar` for checking whether some class contains a certain instance variable and a predicate `inheritance` for checking whether two classes are in an inheritance relationship. Based on predicates such as these, the base layer defines some predicates for reasoning about more complex structural relationships.

- **Transitive closure.** A first way to define more complex relationships from more primitive ones is by computing the transitive closure of those more primitive relationships. A concrete example of this is the `hierarchy` predicate which is the transitive closure of the more primitive `inheritance` predicate.

  To compute the transitive closure of binary relationships, the base layer provides a generic second-order logic predicate *`closure(Relation,Term1,Term2)`*. It takes a binary predicate `Relation` as argument, as well as a start expression `Term1` and an end expression `Term2`, and checks whether `Relation` holds directly or transitively between the two expressions `Term1` and `Term2`. (The predicate is defined in such a way that duplicate results and infinite loops are avoided.)

- **Smalltalk scoping rules.** The predicate `instVarFlattened(Cl,IV)` generalizes the more primitive predicate `instVar`. `instVar(Cl,IV)` merely checks whether a certain class `Cl` contains a certain instance variable `IV`. Due to the scoping rules of Smalltalk, however, all instance variables that belong to a superclass of some class are also visible to that class itself.

[4] To avoid having multiple solutions, we could turn them into a single solution by computing the common superclass of all candidate classes. Of course, if two classes are in separate inheritance hierarchies, the only common superclass may be `Object`.

Therefore, the predicate `instVarFlattened` takes the entire hierarchy into account to check whether a certain instance variable `IV` is visible in some class `Cl`.

Similarly, `understands(Cl,Msg)` generalizes the predicate `classImplementsMethodNamed` to compute all messages that can be understood by (instances of) some class. That is, all messages corresponding to a method that is implemented by this class or one of its superclasses, with the exclusion of all superclass methods that have been cancelled. (There was no such exclusion for instance variables: instance variables cannot be cancelled.) Because the `understands` predicate is used so often (for example, when doing type inferencing) but is rather computationally intensive, the results of this predicate are cached persistently.

As the predicates in the base layer are more high-level than the predicates in the representational layer, we expect them to be less Smalltalk-dependent. Unfortunately, this is not always the case. For example, the typing predicates are Smalltalk-specific because they may provide multiple candidate results. For statically typed object-oriented languages, we would probably prefer variants of these typing predicates that provide a unique result[5]. E.g., a predicate `instVarType` instead of `instVarTypes` and a predicate `hasType` instead of `mayHaveType`.

The predicates that implement the high-level structural relationships are relevant for other object-oriented languages as well. However, their implementation may sometimes be slightly different for those other languages because of possible differences in scoping rules. For example, for Smalltalk, the `understands` predicate needs to take into account that methods may be cancelled. Method cancellation is not supported by many object-oriented languages, though. For this particular example, it may suffice just to replace the predicate that checks for cancelled methods by one that always fails, indicating that no methods can be cancelled in those other languages. (In Subsection 5.3.5 we will show how to check for cancelled methods in the Smalltalk language.)

The predicates that rely on method parse-tree traversal are also relevant for other object-oriented languages, but again, their implementation may be somewhat different, due to differences in the parse-tree representation for those other languages. Most changes, however, will need to be made only to the generic *`traverseMethodParseTree`* predicate and not to the predicates that are defined in terms of this one.

Finally, as in the representational layer, the base layer may also contain predicates that reason about Smalltalk-specific language constructs. Obviously, these predicates are highly Smalltalk-specific. When porting the declarative framework to another object-oriented language, these predicates become obsolete and extra predicates are needed for reasoning about constructs specific to that language. Of course, this remark is relevant for all other layers of the DFW as well.

Coding conventions layer

There is such a thing as a 'Smalltalk culture' which makes that Smalltalk programmers use a lot of widespread conventions [5, 22] to express important intentions for which no explicit language constructs are available. Because of this, for a language like Smalltalk, when declaring architectural mappings we often make use of naming or coding conventions, programming idioms, programming or design styles, design patterns, and so on. Also, due to the absence of (explicit and static) type information in Smalltalk, it is sometimes difficult to express the kinds of architectural mappings we need. Expressing them in terms of conventions often provides a convenient alternative.

A problem with using conventions (or rather, with defining architectural abstractions based on conventions) is that it cannot be guaranteed that the conventions will always be followed in a consistent manner. When some conventions would no longer be followed, some architectural abstractions might produce incorrect results. It is important to see things in the right perspective, though.

First of all, Smalltalk programmers tend to respect the conventions that are part of their culture. Secondly, if a programmer knows that respecting the conventions is important for correct

[5] But even for statically typed object-oriented languages, finding a unique result is not always possible if the language supports type casts.

architectural conformance checking, he or she may be more motivated and disciplined to respect the conventions. Thirdly, we are not forced to define architectural mappings in terms of conventions. Whenever possible, we prefer more precise descriptions based on some kind of semantic inferencing. Unfortunately, sometimes such descriptions require extensive source-code analysis. For example, if we have difficulties describing something due to the lack of type information, we could do a kind of type inferencing (using the typing predicates provided by the base layer). Such a description, however, will probably be more complex and computationally intensive than a description based on conventions. Finally, to aid the programmers in using and respecting the conventions, some language environment support could be offered.

We will come back to most of these issues later in the dissertation (Subsections 7.1.7, 8.3.4 and 8.4.1). In this subsection we merely describe the 'coding conventions layer' which contains many predicates that codify typical Smalltalk coding conventions and styles. Here, we only mention the conventions of which we encountered occurrences in our particular case study (although the conventions themselves are case-independent). Table 5.6 summarizes the corresponding logic predicates that codify these conventions. Based on K. Beck's book on Smalltalk best practice patterns [5], Appendix B provides a more exhaustive list of relevant coding conventions and styles for the Smalltalk language.

Predicate name and arguments	**Meaning of the predicate**
Naming conventions	
`patternMatch(N,P)`	artifact name `N` matches pattern `P`
`stringStartsWith(S1,S2)`	string `S1` starts with substring `S2`
`stringEndsWith(S1,S2)`	string `S1` ends with substring `S2`
`stringContains(S1,S2)`	string `S1` contains substring `S2`
`stringSplit(S1,S2,S3,S4)`	split up string `S1` into substrings `S3` and `S4` occurring before and after substring `S2`
...	...
Coding idioms for methods	
`findMethod(C,M,P)`	method `M` of class `C` matches pattern `P`
`abstractMethod(C,M)`	method `M` of class `C` is abstract
`cancelledMethod(C,M)`	method `M` of class `C` is cancelled
`mutator(C,M,V)`	method `M` of class `C` is mutator of variable `V`
`mutatorMethod(M)`	method `M` is a mutator method
`accessor(C,M,V)`	method `M` of class `C` is accessor of variable `V`
`accessorMethod(M)`	method `M` is an accessor method
`oneToManyStatement(M,V)`	method `M` implements a one-to-many relationship
`instanceCreationMethod(C,M)`	method `M` in class `C` is an instance-creation method
...	...

Table 5.6: Some predicates provided by the coding convention layer.

Naming conventions. When considering an implementation, a lot of important information on the intentions of developers is implicit in the naming conventions that are adopted [5]. The representational layer already provides some very primitive predicates for extracting or retrieving the names of implementation artifacts. However, these are not sufficient for reasoning about the naming conventions that are used. We need more fine-grained predicates that can reason about names at a sub-string level to check whether a name matches a certain string pattern. To this extent we implemented a generic predicate `patternMatch(Name,Pattern)` which takes two arguments: an artifact `Name` and a string `Pattern` to be matched against that name. The pattern resembles a regular expression and supports wildcards, exact matches, prefix matches, postfix matches, (multiple) substring matches, and logic combinations of more primitive patterns: conjunction, disjunction and negation. Some examples of successful pattern matches are listed below:

```
patternMatch('the lonesome man', exact('the lonesome man'))
patternMatch('the lonesome man', contains('lonesome'))
patternMatch('the lonesome man', pattern([_,'lone','some',_,'man',_]))
patternMatch('the lonesome man', and(prefix('the'),postfix('man'),contains('lonesome')))
patternMatch('the lonesome man', or(prefix('a'),prefix('the ')))
patternMatch('the lonesome man', not(contains('x')))
```

This general pattern-match predicate was defined in terms of some more primitive predicates that handle each of the specific cases. For example, `stringStartsWith` checks whether a string starts with a certain substring, `stringEndsWith` checks whether a string ends with a certain substring, `stringContains` checks whether a string contains a certain substring, `stringSplit` splits up a string into substrings occurring before and after a given substring, etc.

Coding idioms for methods. Based on the generic `patternMatch` predicate, a predicate for lexically analyzing a method can be defined: `findMethod(Class,Method,Pattern)`. This predicate checks whether some `Method` in some `Class` matches some `Pattern`. The pattern will be matched to the string-representation of the method's parse tree. Many useful coding idioms that reason about the structure of a method can be defined using this pattern-match predicate:

- `abstractMethod(Class,Method)` In Smalltalk, abstract methods can be recognized because they send a `subclassResponsibility` self send. In other words, (the string representation of) their parse trees matches the following pattern:

  ```
  or(  exact('[send(variable(self),subclassResponsibility,[])]'),
       exact('[return(send(variable(self),subclassResponsibility,[]))]'))
  ```

- `cancelledMethod(Class,Method)` Whereas abstract methods can be recognized because they make a `subclassResponsibility` self send, in Smalltalk, cancelled methods can be recognized because they make a `shouldNotImplement` self send.

- `mutator(Class,Method,VarName)` Mutator methods are methods that assign a value to some variable. These methods can easily be recognized because they typically have the same name as the variable, appended with a ':'. Furthermore, in their body, they only perform an assignment of a value to that variable.

- `mutatorMethod(Method)` verifies whether `Method` is a mutator method and is defined in terms of `mutator(Class,Method,VarName)`.

- `accessor(Class,Method,VarName)` Accessor methods retrieve the value of some variable. They typically have the same name as the variable. Simple accessors do nothing more than returning the value of that variable. (Lazy accessor methods use lazy initialization, which is also characterized by a typical coding idiom. See predicate `lazyInitialisedAccessorMethod` in Appendix B.)

- `accessorMethod(Method)` verifies whether `Method` is an accessor method and is defined in terms of `accessor(Class,Method,VarName)`.

- `oneToManyStatement(Method,InstVar)` In Smalltalk, the typical way to iterate over a collection of elements is to send it an enumerator message (like `do:`, `collect:`, `select:` or `detect:`), pass it a block with one argument representing the element of the collection under consideration, and process that element inside the block by sending the appropriate messages to it. How the results are accumulated or combined depends on the chosen enumerator message. Relying on this coding convention, we can define a predicate `oneToManyStatement(Method,InstVar)` which checks whether some `Method` enumerates over the elements of a collection held in an instance variable `InstVar`.

In addition to analyzing the names or structure of methods, Smalltalk method protocols can provide very useful information on the methods that belong to some protocol. For example, it is a commonly accepted Smalltalk convention to put all instance-creation methods of a class in the same method protocol named '`instance creation`' on the meta class. Using this convention, it is very easy to define a predicate `instanceCreationMethod(Class,Method)` which verifies whether some method is an instance-creation method.

Many other coding conventions can be codified. For more examples of how to codify coding conventions by means of logic predicates we refer to Appendix B and references [54, 86, 87].

Regarding its dependence on the underlying implementation language, the situation for the coding conventions layer is similar to that of the base layer. Some predicates, such as the predicates for string pattern matching, are entirely independent of the chosen implementation language. Other predicates have a Smalltalk-specific implementation, e.g. `abstractMethod` and `instanceCreationMethod`, but could be redefined for other object-oriented languages as well. Yet others, e.g. `cancelledMethod`, are about specific properties or constructs of the Smalltalk language and are therefore of little use for other languages.

Design patterns layer

Based on the more primitive predicates that were declared in the previous layers, we can define some logic predicates that express the structure of design patterns. Examples of such predicates are given in [86, 87]: a rule defining the structure of the Composite design pattern is worked out in [86]; [87] shows how to express the Visitor design pattern; and in the context of his Ph.D. research, R. Wuyts declared some other design patterns as well: the Abstract Factory, Factory Method, Singleton and Bridge pattern. In this subsection, we mention only two examples which we encountered in our case study: the Composite and Factory Method design patterns.

Predicate name and arguments	Meaning of the predicate
`compositePattern(A,C,M)`	Abstract class `A` and composite class `C` conform to the structure of the Composite design pattern
`factoryMethod(C,M)`	Method `M` is a Factory Method for class `C`
...	...

Table 5.7: Some predicates provided by the design patterns layer.

Composite. We explain the Composite design pattern by means of a concrete example. The Smalltalk implementation of SOUL contains an instance of the Composite design pattern in the class hierarchy representing logic terms. Different kinds of logic terms can be distinguished: variables, constants, functors and term sequences. Term sequences are a special kind of terms which represent a composition of other terms. This is indeed an occurrence of the Composite pattern. The key to this pattern is an abstract class which represents both primitives and their containers. For the term hierarchy, there is such a class: every kind of term is represented by a class which inherits from an abstract superclass `SOULAbstractTerm`. In particular, term sequences are represented by a class `SOULTerms` which inherits from that superclass. The class `SOULTerms` contains an instance variable representing a collection of terms. As required by the abstract superclass, it also implements a set of typical operations on terms such as interpretation and the substitution of bindings. These operations on `SOULTerms` are implemented by methods which recursively invoke the same method on each of the terms it contains and then combine the returned results in the appropriate way. For more details on the Composite design pattern, we refer to [23, p. 163].

This structural relationship between the composite class (in our example, `SOULTerms`) and the abstract superclass (in our example, `SOULAbstractTerm`) can easily be codified in a logic rule

```
compositePattern(Abstract, Composite, Message)
```

The third argument is optional and represents the name of the method on the composite class that is recursively called on the components it contains.

Factory Method. We also explain the Factory Method design pattern on the basis of a concrete example. In the Smalltalk implementation of SOUL, logic repositories are represented by the abstract class `SOULAbstractRepository` or one of its subclasses. To provide some flexibility regarding the kinds of repositories that may be created, the SOUL developers avoided to state the names of classes representing repositories explicitly into the code. Instead, a Factory Method design pattern is used to create new instances of SOUL repositories. To this extent, a factory class `SOULFactory` is defined which implements, amongst others, some methods to create repositories. These methods are called Factory Methods. (The same factory class also contains some Factory Methods for other SOUL-specific classes.) In order to instantiate the repository classes, these creation methods directly refer to them. In all places in SOUL where repositories need to be created, this is done indirectly by calling one of these Factory Methods, instead of directly invoking an instance-creation method on a repository class.

The Factory Method pattern can easily be codified by means of a logic predicate

```
factoryMethod(Class, Method)
```

A Factory Method is merely a `Method` which does nothing more than directly sending an instance-creation message to some `Class`. This predicate can be used, amongst others, to find every potential Factory `Method` for some `Class`, by verifying whether the method matches the pattern. For more details on the Factory Method pattern, we refer to [23, p. 107].

The predicates in the design patterns layer express high-level design structures that are relevant for most object-oriented languages. Although the concrete implementation of a design pattern may depend on the chosen implementation language, the structure is largely language-independent. Therefore, the same predicates (but possibly with a slightly different implementation) may be relevant for other object-oriented languages.

5.3.6 The architectural layer of the DFW

Finally, we turn our attention to the architectural layer of the declarative framework. Currently, this layer contains only one sub-layer. Essentially, this sub-layer defines a whole range of predicates that capture the most common ways of mapping architectural abstractions to implementation artifacts and their dependencies.

Architectural mapping predicates

We structure our discussion of the predicates in this layer according to the different kinds of architectural abstractions. For each kind we discuss which predefined predicates are useful to define typical architectural mappings for those architectural abstractions. As before, some of these predicates are largely independent of the chosen implementation language, whereas some others are Smalltalk-specific.

Virtual classifications. We repeat that virtual classifications are computed sets of implementation artifacts. Such virtual classifications can be defined in many ways:

1. Directly, in terms of more primitive predicates defined by the lower-level layers:
   - A predicate like `findMethod` (see coding conventions layer) can be used to find all methods that match a certain pattern. Similar predicates can be defined in terms of the `patternMatch` predicate to find all artifacts of other kinds (e.g., classes or instance variables) that match a certain pattern.

- Some predicates of the implementation layer that reason about classes can be used to compute a classification consisting of classes. For example, we can use:
  - `classInCategory` to compute all classes that belong to some Smalltalk class category;
  - `hierarchy` to compute all classes that belong to the hierarchy of some root class;
  - `classImplementsMethodNamed` to compute all classes that implement a method with a certain name;
  - `methodInProtocol` to compute all classes that have at least one method in a given method protocol.
- Some predicates of the implementation layer that reason about methods can be used to compute a classification consisting of methods. For example, we can use:
  - `classImplementsMethod` to compute all methods belonging to some class;
  - `methodInProtocol` to compute all methods that belong to a given method protocol.
- Some predicates of the implementation layer that reason about instance variables can be used to compute a classification consisting of instance variables. For example, we can use:
  - `instVar` to compute all instance variables that belong to some class;
  - `instVarFlattened` to compute all instance variables that belong to some class or one of its superclasses.

2. In terms of an already declared virtual classification. This can either be done directly in terms of the primitive predicate `classifiedAs(ClassificationID,Artifact)` which checks whether a certain artifact belongs to a specified classification, or we can use a more high-level predicate like

   ```
   findClassesFromMethods(Class, ClassificationID)
   ```

   which can be used to compute all classes that implement a method belonging to some specified classification. A similar predicate

   ```
   findMethodsFromClasses(Method, ClassificationID)
   ```

   can be used to compute all methods that belong to a class in some specified classification. Many other similar predicates can be defined.

3. By combining already defined classifications with operators such as `union`, `intersection` and `difference`.

4. In terms of high-level dependencies among implementation artifacts. For example, we can use:

   - `mentions_M_M` to compute all methods that explicitly mention (or are mentioned by) a certain method;
   - `invokes_M_M` to compute all methods that invoke (or are invoked by) a given method;
   - `createsInstanceOf_C_C` to find all classes that create an instance of (or are created by) some other class.

All these predicates are summarized in Table 5.8, according to the same four categories.

Predicate name and arguments	Meaning of the predicate
Lower-level predicates in terms of which to define virtual classifications	
findMethod(C,M,P)	method M of class C matches pattern P
classInCategory(Ca,Cl)	class Cl belongs to class category Ca
hierarchy(Root,Cl)	class Cl belongs to the hierarchy of class Root
classImplementsMethod(Cl,Me)	method Me belongs to class Cl
classImplementsMethodNamed(Cl,MN,_)	class Cl implements a method with name MN
methodInProtocol(Cl,Pr,_)	class Cl has at least one method in protocol Pr
methodInProtocol(_,Pr,Me)	method Me belongs to method protocol Pr
instVar(Cl,IV)	instance variable IV belongs to class Cl
instVarFlattened(Cl,IV)	instance variable IV belongs to class Cl or a superclass
...	...
Predicates that compute virtual classifications from already defined ones	
classifiedAs(C,A)	artifact A belongs to classification C
findClassesFromMethods(Cl,C	class Cl implements a method belonging to classification C
findMethodsFromClasses(Me,C)	method Me belongs to a class in classification C
findMetaClassesFromClasses(MC,C)	MC is meta class of a class in classification C
...	...
Predicates that implement operators on virtual classifications	
union(C1,C2,A)	artifact belongs to union of 2 classifications
intersection(C1,C2,A)	artifact belongs to intersection of 2 classifications
difference(C1,C2,A)	artifact belongs to difference of 2 classifications
...	...
High-level dependencies among implementation artifacts	
asks_M_M(M1,M2)	method M1 invokes method M2 and uses the returned result afterwards
asks_C_M(C,M)	some method of class C invokes method M and uses the returned result afterwards
uses_C_M(C,M)	class C has a method which uses method M
uses_M_E(M,E)	expression E is used somewhere inside the body of method M
invokes_M_M(M1,M2)	method M1 invokes method M2
invokesMutator_M_M(M1,M2)	method M1 invokes a (direct or indirect) mutator method M2
specializes_C_C(C1,C2)	class C1 is specialization of class C2
specializes_M_M(M1,M2)	method M1 is specialization of method M2
concretizes_M_M(M1,M2)	method M1 is concretization of method M2
hasPart_C_C(C1,C2)	class C1 has instance of class C2 as part
isComposite_C_C(C1,C2)	class C1 is special kind of class C2 and contains instances of class C2
hasParameterType_M_C(M,C)	method M has argument of type C
mentions_M_M(M1,M2)	method M1 explicitly mentions the name of method M2 in its body
createsInstanceOf_M_C(M,C)	method M creates instance of class C
createsInstanceOf_C_C(C1,C2)	class C1 creates instance of class C2
...	...

Table 5.8: Some architectural mapping predicates for defining virtual classifications and virtual dependencies.

Virtual dependencies. The last category of predicates in Table 5.8 describes high-level dependencies among implementation artifacts. These predicates cannot only be used to define virtual classifications (as explained above), but also for defining virtual dependencies. Virtual dependencies can represent simple implementation relationships that can be derived almost directly from the source code such as message invocation (`invokes_M_M`) or inheritance (`specializes_C_C`). More complex relationships such as class instantiation (`createsInstanceOf_M_C`) may depend on the use of certain coding conventions (e.g., instance-creation methods belong to the 'instance creation' method protocol of a class) and design patterns.

Filters. The most frequently-used filters are those which accept only artifacts of a certain kind (i.e., base classes, meta classes, methods, instance variables, etc.). For example, `baseClassFilter` accepts only base (i.e. non-meta) Smalltalk classes and a `methodFilter` succeeds only for Smalltalk methods. The name of these filters suggest the kind of artifacts they accept. There are also two trivial filters: `identityFilter` is the trivial filter which accepts all artifacts and `forgetfulFilter` is the trivial filter which rejects all artifacts. Table 5.9 lists all these filters.

Predicate name and arguments	Meaning of the predicate
`identityFilter(Artifact)`	accepts any `Artifact` (always succeeds)
`forgetfulFilter(Artifact)`	accepts nothing (always fails)
`classFilter(Artifact)`	accepts only classes (either base or meta classes)
`baseClassFilter(Artifact)`	accepts only base classes
`metaClassFilter(Artifact)`	accepts only meta classes
`methodFilter(Artifact)`	accepts only methods
`instVarFilter(Artifact)`	accepts only instance variables
...	...

Table 5.9: Some architectural mapping predicates representing predefined filter predicates.

Ports that represent actions or processes are typically mapped to method filters. Ports that represent types are typically mapped to class filters. Ports that represent data are often mapped to instance variable filters. In addition to these general predefined filters, it is possible for an architect to define his or her own domain-specific filters. An example of this will be given in Section 7.2.

Given a filter and a virtual classification, the following second-order logic predicate can be used to generate artifacts that belong to the classification and satisfy the filter:

```
filteredIsClassifiedAs(Classification, Filter, Artifact) :-
    classifiedAs(Classification, Artifact),
    Filter(Artifact).
```

Quantifiers. Quantifiers specify how to generalize a relationship among artifacts to a relationship among sets of artifacts. The DFW provides predefined predicates representing the set quantifiers $\forall$ and $\exists$ as well as some special versions of these predicates which report special information to the user in case of failure. Table 5.10 summarizes these predefined quantifier predicates. All these quantifier predicates are very general second-order predicates that are independent of the chosen implementation language.

In our experiments we only used the $\forall$ and $\exists$ quantifiers. Therefore, they are the only ones that have been implemented in the DFW (as well as their 'debug' versions). In addition to these predefined quantifier predicates, other useful examples of quantifier predicates are:

- `existsUnique` applies the `Test` predicate one by one to each of the generated values. The application should succeed for *exactly one* of the generated values.

Predicate name and arguments	**Meaning of the predicate**
`forall(Generator,Test)`	does `Test(X)` hold $\forall$ `X` that satisfy `Generator(X)`
`exists(Generator,Test)`	$\exists$ `X` that satisfies `Generator(X)` and `Test(X)`
`forallDebugOne(Generator,Test)`	same as `forall`, but reports first failure on console
`forallDebugAll(Generator,Test)`	same as `forall`, but reports all failures on console
`existsDebug(Generator,Test)`	same as `exists`, but reports all generated values on console in case of failure
...	...

Table 5.10: Some architectural mapping predicates representing predefined quantifier predicates.

- Other predicates could be imagined that represent various quantifiers, cardinalities or multiplicities (2 or more, between 3 and 5, less than 4, more than half, ...), UML qualifiers or qualified associations, and so on.

5.4 Formal definitions

In this section we formalize our approach to architectural conformance checking in four steps:

1. we formalize the ADL by defining the structure of well-formed conceptual architectures (Subsection 5.4.2);
2. we formalize the architectural abstraction language by defining the different kinds of architectural abstractions in terms of implementation artifacts (Subsection 5.4.3);
3. we formalize the architectural instantiation language by defining mapping functions from ADL constructs to architectural abstractions (Subsection 5.4.4);
4. we formalize the conformance checking algorithm by defining a denotational semantics which maps conceptual architectures to Boolean values (Subsection 5.4.5).

The purpose of the formalization is to define the semantics of a conceptual architecture A in terms of the implementation I to which it is mapped. Because we are merely interested in the result of performing a conformance check and not in how it is actually achieved, we use a denotational semantics [59]. The semantics of a conceptual architecture is a truth value which indicates whether or not the implementation conforms to the architecture.[6] Before defining this semantic function and the domains on which it operates, we present some notations that are needed in our formalization.

5.4.1 Notations

Total function.
The domain of all total functions from A to B is denoted as $A \rightarrow B$.

Whereas a total function $f : A \rightarrow B$ maps every element of A to an element of B, partial functions may be undefined for some elements of the function domain A.

Partial Function.
The domain of all partial functions from A to B is denoted as $A \hookrightarrow B$.

Because total functions are a special case of partial functions, sometimes we simply use the term 'function' when we actually mean 'partial function'.

Domain and range.
Let $f : A \hookrightarrow B$;
The function domain of f is defined as: $dom(f) = \{\ a \in A \mid \exists\ b \in B : f(a) = b\ \}$
The range of f is defined as: $range(f) = \{\ b \in B \mid \exists\ a \in A : f(a) = b\ \}$

Note that for a total function $f : A \rightarrow B$, $dom(f) = A$.

Injective, surjective and bijective functions.

A function $f : A \hookrightarrow B$ is injective	$\Leftrightarrow$	$\forall\ a_1, a_2 \in A : f(a_1) = f(a_2) \Rightarrow a_1 = a_2$.
A function $f : A \hookrightarrow B$ is surjective	$\Leftrightarrow$	$range(f) = B$.
A function $f : A \hookrightarrow B$ is bijective	$\Leftrightarrow$	*f is injective and f is surjective.*

A function is said to be *finite* if its domain is finite.

[6] In an industrial-strength tool, we would like to have some more information on the result of a conformance check. More precisely, when conformance checking fails (i.e., when the semantic function returns false), we would like to know where and why the conformance conflict occurred. We will come back to this issue in Section 7.4, where we show how the conformance checking algorithm can be enhanced to generate such information.

Finite Function.
The domain of all finite partial functions from A to B is defined as:
$A \hookrightarrow_{fin} B \ = \{\ f : A \hookrightarrow B \mid dom(f) \text{ is finite } \}$

Note that, if $f \in A \hookrightarrow_{fin} B$ then $range(f)$ is also finite.

The *restriction* of a function to a subset of its function domain is defined as follows:

Restriction.
Let $f : A \hookrightarrow B$ and $D \subseteq A$;
$f_{|D} : D \hookrightarrow B : d \mapsto f(d)$

In addition to these notations for denoting functions, we need the following notations for denoting the powerset of some set (i.e., the set of all subsets of some set).

Powerset.
The powerset of a set A is denoted and defined as:
$\mathcal{P}(A) \ = \{\ K \mid K \subseteq A\ \}$

The finite powerset of a set A is the set of all finite subsets of A:
$\mathcal{P}_{fin}(A) \ = \{\ K \mid K \text{ is finite and } K \subseteq A\ \}$

$\mathbb{N}$ denotes the set of all natural numbers (i.e., positive integers) and $\mathbb{N}_0$ is the set of all non-zero natural numbers.

Natural numbers.
$\mathbb{N} \ = \{0, 1, 2, 3, \ldots\}$
$\mathbb{N}_0 \ = \mathbb{N} \setminus \{0\} = \{1, 2, 3, \ldots\}$

Finally, the symbol $+$ represents string concatenation and $\oplus$ generalizes this notation over domains of strings. More precisely, $A \oplus B$ is the domain of all strings which are a concatenation of a string in A with a string in B.

Sring concatenation.
$+ : String \times String \rightarrow String : (a, b) \mapsto s$ *such that s is the string concatenation of a and b.*
$\oplus : \mathcal{P}(String) \times \mathcal{P}(String) \rightarrow \mathcal{P}(String) : (A, B) \mapsto \{\ a + b \mid a \in A \ \wedge \ b \in B\ \}$

5.4.2 Formalizing the architecture description language

We formalize the ADL by defining domains[7] representing the different parts of a conceptual architecture and by defining well-formedness constraints on the elements of those domains.

ADL Syntax

To simplify the formalization, we assume that every entity in a conceptual architecture (i.e., an architectural view, concept, relation, port or role) is uniquely identified by its name. This assumption is somewhat over-restrictive, though. It would suffice to assume that every entity has a name which is unique in its scope (as we did in Section 5.2). For example, it should be allowed for two ports to have the same name, as long as they belong to a different concept. (For example, in Figure 4.4 on page 46, both the concept **User Application** and **Auxiliary Application** have a port named Type, but this causes no confusion as the concepts themselves have different names.) In such cases a unique name can always be constructed by appending the name of the nested

[7] We describe the syntax in terms of domains, rather than using BNF or some other notation, because we need these domains when defining our denotational semantics.

entity to the unique name of the entity in which it is nested. (Applying this to Figure 4.4 would mean that we should qualify the concepts named 'User Application' and 'Auxiliary Application' with the prefix 'User Interaction.' representing the name of the architectural view. The 'Type' port of each of these concepts should be prefixed with 'User Interaction.User Application.' or 'User Interaction.Auxiliary Application.' representing the unique name of the concept.) Hence, we adopt the following naming convention, where *String* denotes the set of all possible alphanumerical character strings.

Names.

$ArchName \cup ViewName \cup ConceptName \cup RelName \cup PortName \cup RoleName \subset String$

$ArchName \cap ViewName \cap ConceptName \cap RelName \cap PortName \cap RoleName = \emptyset$

As explained in Section 5.2, we define a conceptual architecture as a name space of architectural views. This name space is finite: intuitively, a conceptual architecture consists of a finite set of architectural views. A conceptual architecture also has a unique name. This is expressed by the injectiveness constraint on $ArchName$.

Conceptual architecture.

The domain of all conceptual architectures is:

$Architecture = ArchName \times (ViewName \hookrightarrow_{fin} View)$

where the first projection function is injective on Architecture.

For a given conceptual architecture $A \in Architecture$, $archName(A)$ returns the name of that conceptual architecture; $views(A)$ denotes the (finite) set of all architectural views belonging to A; $view_A(n)$ returns the unique architectural view with name n in A.

Architecture selectors.

$archName : Architecture \hookrightarrow ArchName : (N, f) \mapsto N$

$views : Architecture \hookrightarrow \mathcal{P}_{fin}(View) : (N, f) \mapsto range(f)$

Let $A = (N, f) \in Architecture$;

$view_A : ViewName \hookrightarrow_{fin} View : n \mapsto f(n)$

An architectural view consists of a finite set of concepts, a finite set of relations, and a finite set of links between them.

Architectural view.

The domain of all architectural views is:

$View = \mathcal{P}_{fin}(Concept) \times \mathcal{P}_{fin}(Relation) \times \mathcal{P}_{fin}(Link)$

Of course, the links in an architectural view should connect only (ports of) concepts and (roles of) relations that belong to that architectural view. Later on, we will express this as a well-formedness constraint on architectural views.

We use the notations $concepts(V)$, $relations(V)$ and $links(V)$ to retrieve the different parts of some architectural view $V \in View$. They correspond to the first, second and third projection functions.

View selectors.

$concepts$	:	$View \to \mathcal{P}_{fin}(Concept)$	:	$(C, R, L) \mapsto C$
$relations$	:	$View \to \mathcal{P}_{fin}(Relation)$	:	$(C, R, L) \mapsto R$
$links$	:	$View \to \mathcal{P}_{fin}(Link)$	:	$(C, R, L) \mapsto L$

In an architectural view, every concept has a unique name and a finite set of ports defining the external interface of that concept. Uniqueness of the name is again expressed by an injectivity constraint. Relations are similar to concepts, the only difference being that they contain a set of roles instead a set of ports.

Concepts.
$Concept = ConceptName \times \mathcal{P}_{fin}(Port)$
where $ConceptName = ArchName \oplus \{'.'\} \oplus ViewName \oplus \{'.'\} \oplus LocConcName$
and $LocConcName \subset String$
and *the first projection function is injective on* $Concept$.

Relations.
$Relation = RelName \times \mathcal{P}_{fin}(Role)$
where $RelName = ArchName \oplus \{'.'\} \oplus ViewName \oplus \{'.'\} \oplus LocRelName$
and $LocRelName \subset String$
and *the first projection function is injective on* $Relation$.

The '.' symbol is used as part of a string to denote the nesting of concepts or relations in architectural views. Note that the definition $ConceptName = ArchName \oplus \{'.'\} \oplus ViewName \oplus \{'.'\} \oplus LocConcName$ does not conflict with the earlier restrictions put on $ConceptName$ when we defined the naming conventions. Instead, it refines those earlier restrictions with an extra constraint. Concept names are globally unique and local concept names are unique with respect to an architectural view. The same remark holds for relation names.

We use the notations $concName(c)$, $relName(r)$, $ports(c)$ and $roles(r)$ to retrieve the different parts of some concept c or relation r. Again, these selector functions are mere projections.

Element selectors.

$concName:$	$Concept \to ConceptName:$	$(n, P) \mapsto n$
$ports:$	$Concept \to \mathcal{P}_{fin}(Port):$	$(n, P) \mapsto P$
$relName:$	$Relation \to RelName:$	$(n, R) \mapsto n$
$roles:$	$Relation \to \mathcal{P}_{fin}(Role):$	$(n, R) \mapsto R$

To retrieve a concept or relation with a certain name from some conceptual architecture A, the following functions can be used.

Architecture selectors (2).
Let $A \in Architecture$;

$concept_A : ConceptName \hookrightarrow_{fin} Concept : n \mapsto c$
such that $\exists\, V \in views(A) :\ c \in concepts(V)\ \wedge\ concName(c) = n$

$relation_A : RelName \hookrightarrow_{fin} Relation : n \mapsto c$
such that $\exists\, V \in views(A) :\ r \in relations(V)\ \wedge\ relName(r) = n$

Note that the concept c and relation r in the above definition are uniquely defined, because every concept and relation have a unique name. Also note that both selector functions are finite partial functions. Partial because not every possible concept name (resp. relation name) necessarily has a concept (resp. relation) assigned to it. Finite because every conceptual architecture has a finite set of architectural views and every architectural view contains a finite set of concepts and relations.

If we are interested in knowing all concepts or relations that belong to some conceptual architecture A, we can use the following shortcuts:

Architecture shortcuts.
Let $A \in Architecture$;

$Concepts_A = range(concept_A)$
$Relations_A = range(relation_A)$

Note that $Concepts_A \in \mathcal{P}_{fin}(Concept)$ and $Relations_A \in \mathcal{P}_{fin}(Relation)$.

Ports and roles are characterized by their name only, which is a concatenation of their local name and the name of the architectural element in which they are nested.

Ports.
$Port = PortName$
where $PortName = ConceptName \oplus \{'.'\} \oplus LocPortName$
and $LocPortName = String$

Ports.
$Role = RoleName$
where $RoleName = RelName \oplus \{'.'\} \oplus LocRoleName$
and $LocRoleName = String$

Similar to the construction of names for concepts and relations, the name of a port is constructed by appending its local name to the name of the concept to which it belongs. Again, we note that this definition only refines our earlier naming conventions, and that port names are globally unique and local port names are unique with respect to the concept to which they belong. The same remark holds for role names.

To retrieve the names of concept ports or relation roles, we use the functions *portName* and *roleName* which are mere identity functions.

Port and role selectors.
$portName: \quad Port \to PortName: \quad p \mapsto p$
$roleName: \quad Role \to RoleName: \quad r \mapsto r$

Finally, we define links between ports and roles as couples that associate a port with a role.

Links.
$Link = Port \times Role$

Note that a link is uniquely defined by a port and a role. In other words, there can be only one link between a port and a role. However, it is allowed for multiple roles to be associated with the same port and vice versa (via different links). Also note that links have no direction associated with them. They just connect a port to a role.

We will use the projection functions $port(l)$ and $role(l)$ to retrieve the different parts of some link $l \in Link$.

Link selectors.
$port: \quad Link \to Port: \quad (p, r) \mapsto p$
$role: \quad Link \to Role: \quad (p, r) \mapsto r$

To find a port or role with a certain name in some conceptual architecture A, or to find the (unique) link between some port and role, we can use the functions below. The functions are well-defined, because the definitions of ports and roles imply that there can be only one port or role associated with a certain name.

Architecture selectors (3).
Let $A \in Architecture$;

$port_A : PortName \hookrightarrow_{fin} Port : n \mapsto p$
such that $\exists\, V \in views(A) : \exists\, c \in concepts(V) : p \in ports(c) \ \wedge\ portName(p) = n$

$role_A : RoleName \hookrightarrow_{fin} Role : n \mapsto r$
such that $\exists\, V \in views(A) : \exists\, rel \in relations(V) : r \in roles(rel) \ \wedge\ roleName(r) = n$

$link_A : Port \times Role \hookrightarrow_{fin} Link : (p, r) \mapsto l = (p, r)$
such that $\exists\, V \in views(A) : \ l \in links(V)$

Note again that both selector functions $port_A$ and $role_A$ are finite partial functions. Partial because not every possible port name (resp. role name) necessarily has a port (resp. role) in A assigned to it. Finite because every conceptual architecture has a finite set of architectural views, every architectural view contains a finite set of concepts (resp. relations), and every concept (resp. relation) has a finite set of ports (resp. roles). The selector function $link_A$ is also a finite partial function. Finite because A has a finite set of architectural views, each with a finite set of links. Partial because not every port and role are linked.

If we are interested in knowing all ports, roles or links that belong to some conceptual architecture A, we can use the following shortcuts:

Architecture shortcuts (2).
Let $A \in Architecture$;

$Ports_A = range(port_A)$
$Roles_A = range(role_A)$
$Links_A = range(link_A)$

Note that $Ports_A \in \mathcal{P}_{fin}(Port)$, $Roles_A \in \mathcal{P}_{fin}(Role)$ and $Links_A \in \mathcal{P}_{fin}(Link)$.

Well-formedness of the ADL

For the above definitions to be well-formed, some extra constraints need to be satisfied. First of all, an architectural view is well-formed if there exists no other architectural view with the same name in the conceptual architecture (which is always satisfied, thanks to the definition of a conceptual architecture) and if the links for that architectural view only mention ports of concepts and roles of relations that belong to the same view.

Well-formed architectural view.
Let $A \in Architecture$ *and* $V \in views(A)$;

V *is well-formed if* $\forall\, l \in links(V)$:
$(\ \exists\, c \in concepts(V) : port(l) \in ports(c)\ )\ \wedge\ (\ \exists\, r \in relations(V) : role(l) \in roles(r)\ )$

A concept (resp., relation) is well-formed if its name is consistent with the view and conceptual architecture to which it belongs. In other words, the view name and architecture name mentioned as a prefix in the concept's (resp., relation's) full name must be the one of the view and architecture in which it is nested.

Well-formed concept.
Let $A \in Architecture$ *and* $c \in Concept$;

c *is well-formed in* A *if*
$concName(c) = archName(A) + '.' + viewName + '.' + locConcName$
where $viewName \in ViewName$ *and* $c \in concepts(view_A(viewName))$
and $locConcName \in LocConcName$

Well-formed relation.
Let $A \in Architecture$ *and* $r \in Relation$;

r *is well-formed in* A *if*
$relName(c) = archName(A) + '.' + viewName + '.' + locRelName$
where $viewName \in ViewName$ *and* $r \in relations(view_A(viewName))$
and $locRelName \in LocRelName$

Similarly, a port is well-formed if its name is consistent with the name of the concept to which it belongs. In other words, the concept name mentioned in the port's full name must be the one of the concept in which the port is nested.

Well-formed concept port.
Let $A \in Architecture$ *and* $p \in Port$;

p *is well-formed in* A *if*
$portName(p) = concName + '.' + locPortName$
where $concName \in ConceptName$ *and* $p \in ports(concept_A(concName))$
and $locPortName \in LocPortName$

Note that in the well-formedness constraint for ports, we do not need to check the view and architecture in which the port occurs, as this is already checked by the well-formedness constraint for concepts.

The well-formedness constraint for roles is very similar.

Well-formed relation role.
Let $A \in Architecture$ *and* $r \in Role$;

$r \in Role$ *is well-formed in* A *if*
$roleName(r) = relName + '.' + locRoleName$
where $relName \in RelName$ *and* $r \in roles(relation_A(relName))$
and $locRoleName \in LocRoleName$

For the above well-formedness constraints to be unambiguously defined, we must assume that view names and local names of concepts, relations, ports or roles are not allowed to contain the symbol '.'. Otherwise, there may be multiple solutions for the name equality. For example, consider the equality $concName(c) = archName + '.' + viewName + '.' + localConceptName$. If the concept c would have name 'a.b.c.d', it is not clear whether the name of the conceptual architecture would be 'a' or 'a.b'; whether the name of the view would be 'b', 'c', or even 'b.c'; and so on. To solve this problem we need to impose an additional well-formedness constraints on names: for local names, view names and names of conceptual architectures it is not allowed to use strings that contain the symbol '.'. $NoDotString$ denotes the set of all alphanumeric character strings that do not contain the character '.'.

Well-formed names.
$ArchName \cup ViewName \cup LocConcName \cup LocRelName \cup LocPortName \cup LocRoleName \subset NoDotString \subset String$

5.4.3 Formalizing the architectural abstraction language

In this subsection, we define the different constructs of the architectural abstraction language: virtual classifications, virtual dependencies, filters, and quantifiers. These constructs are intermediary abstractions to define the mapping of architectural entities to an implementation. To keep things simple, we formalize an implementation I as a set of implementation artifacts in some programming language L.

Implementation artifact.
For some programming language L,
$Artifact_L$ denotes the domain of all possible implementation artifacts in that language L.

We do not give a formal definition of $Artifact_L$, as this strongly depends on the chosen programming language L. In general, an implementation artifact $a \in Artifact_L$ is a well-defined language construct in language L. For example if $L = Smalltalk$ than $Artifact_L$ is the set of all possible classes, methods, instance variables, etc.

An implementation in programming language L is a finite set of implementation artifacts in that language. Of course, not every set of implementation artifacts constitutes a valid implementation. Extra constraints are needed to specify when an implementation is well-formed. We do not formally define these constraints, as they strongly depend on the programming language. One example of such a constraint on Smalltalk implementations is that there can be no two classes with the same name.

Implementation.
The domain of all implementations in some programming language L is defined as:

$$Implementation_L = \mathcal{P}_{fin}(Artifact_L)$$

Note that the above definitions of 'implementation' and 'implementation artifact' are very general. In fact, there is no need to restrict ourselves to implementation artifacts and programming languages. The same definitions would remain valid for, for example, design artifacts and design languages. Our formalism is entirely independent of the chosen base language. However, because the main focus of this dissertation is on checking architectural conformance of an implementation, we prefer to talk about 'implementations' and 'implementation artifacts'.

We formally define a software classification for some implementation I in programming language L as a finite set of implementation artifacts belonging to that implementation. (The set is finite because the implementation itself is already finite.)

Software classification.
The domain of all software classifications for some implementation $I \in Implementation_L$ in some programming language L is defined as:

$$Classification_{I,L} = \mathcal{P}_{fin}(I)$$

We do not make a distinction between virtual and ordinary software classifications. Whereas ordinary classifications are defined extensionally, virtual classifications are declared intentionally and can be computed 'by need'. Of course, we could formalize virtual classifications as functions that compute a set of implementation artifacts upon invocation (rather than defining them directly as a set of implementation artifacts). The main purpose of the formalization, however, is to rigorously define the semantics of a conceptual architecture in terms of the implementation to which it is mapped. Whether or not to represent classifications intentionally is not relevant in such a denotational semantics.

We define filters as functions that take a classification of implementation artifacts as input and return a subset of their input.

Filters.
The domain of all filters for some implementation $I \in Implementation_L$ in some programming language L is defined as:

$$Filter_{I,L} = \{F \in Classification_{I,L} \to Classification_{I,L} \mid \forall\, C \in Classification_{I,L} :\ F(C) \subseteq C\ \}$$

Intuitively, virtual dependencies represent relationships over implementation artifacts in some programming language L. Formally, a relationship is a set of tuples on some domain of values.

Relation.
The domain of all relationships on some domain of values A is defined as:

$Relationship_A = \bigcup_{n \in \mathbb{N}_0} Relationship_{A,n}$ *where* $Relationship_{A,n} = \mathcal{P}(A^n)$

Using the above definition, we formalize a virtual dependency as a relationship on implementation artifacts.

Virtual dependencies.
The domain of all virtual dependencies for some implementation $I \in Implementation_L$ in programming language L is defined as:
$Dependency_{I,L} = Relationship_I$

The domain of all n-ary virtual dependencies for some implementation $I \in Implementation_L$ in programming language L is defined as:
$Dependency^n_{I,L} = Relationship_{I,n}$ *where* $n \in \mathbb{N}_0$

To determine the arity of a virtual dependency, we provide the following function:

Arity.
Let L be some programming language and $I \in Implementation_L$;
$arity : Dependency_{I,L} \to \mathbb{N}_0 : d \mapsto n$ *such that* $d \in Dependency^n_{I,L}$

Finally, we formalize the notion of quantifiers. Intuitively, a quantifier is a second-order function which takes two inputs:

1. a (first-order) function $\lambda x.f(x) \in Artifact_L \to Boolean$ which takes as input an implementation artifact x and returns a Boolean;

2. a set $C \in Classification_{I,L}$ of implementation artifacts.

It applies the function $\lambda x.f(x)$ to each of the implementation artifacts $x \in C$, and combines all returned (Boolean) results to produce a new Boolean value. In general, the domain of all quantifiers can be defined as follows:

Quantifiers.
The domain of all quantifiers for some implementation $I \in Implementation_L$ in some programming language L is defined as:

$Quantifier_{I,L} = (Classification_{I,L} \times (Artifact_L \to Boolean)) \to Boolean$

For example, a quantifier corresponding to '$\forall$' takes the conjunction of all results, and a quantifier corresponding to '$\exists$' takes the disjunction of all results:

Predefined quantifiers.
Let L be some programming language and $I \in Implementation_L$;

$$Forall_{I,L} : (Classification_{I,L} \times (Artifact_L \to Boolean)) \to Boolean : (C, \lambda x.f(x)) \mapsto \bigwedge_{x \in C} f(x)$$

$$Exists_{I,L} : (Classification_{I,L} \times (Artifact_L \to Boolean)) \to Boolean : (C, \lambda x.f(x)) \mapsto \bigvee_{x \in C} f(x)$$

Other quantifiers can be defined in a similar way. For example, a quantifier $ExistsUnique_{I,L}$ corresponding to the quantifier symbol '$\exists!$' takes the exclusive disjunction (i.e., 'xor') of all expressions $f(x)$ for all $x \in C$.

When checking conformance, quantifiers are used to apply virtual dependencies over software classifications. Unfortunately, virtual dependencies are modeled as sets of tuples, whereas quantifiers expect a unary Boolean function as input. Therefore, we need a way of transforming virtual dependencies into such unary Boolean functions. We will do this by translating a relationship R into a 'curried' function $\lambda x_n \ldots \lambda x_2.\lambda x_1.f(x_1, x_2, \ldots, x_n)$ such that $f(x_1, x_2, \ldots, x_n)$ is true if and only if $(x_1, x_2, \ldots, x_n) \in R$. Instead of taking its arguments all at once, a curried function consumes its arguments one by one. Note that we also reverse the order of the arguments, requiring the last argument to be input first; we will explain later why we do this.

Currying.
Let A be some domain of values;

We recursively define the following domains of 'curried' Boolean functions on A:
$CurriedFunction_{A,1} = A \rightarrow Boolean$
$CurriedFunction_{A,n} = A \rightarrow CurriedFunction_{A,n-1} \quad \forall\ n > 1$

The following function transforms relationships into curried Boolean functions:
$curry_{A,n} : Relationship_{A,n} \rightarrow CurriedFunction_{A,n} : R \mapsto \lambda x_n \ldots \lambda x_2.\lambda x_1.f(x_1, x_2, \ldots, x_n)$
where $f(x_1, x_2, \ldots, x_n) = True \quad \Leftrightarrow \quad (x_1, x_2, \ldots, x_n) \in R$

5.4.4 Formalizing the architectural instantiation language

Now that we have formally defined the domains of the ADL and of the architectural abstraction language, we can define the architectural instantiation language as a set of mapping functions from architectural entities (in the ADL) to architectural abstractions (in the architectural abstraction language):

Architectural instantiation.
Let L be some programming language, $I \in Implementation_L$ and $A \in Architecture$;

$$
\begin{array}{lcl}
conceptMapping^A_{I,L} & : & Concept \hookrightarrow_{fin} Classification_{I,L} \\
portMapping^A_{I,L} & : & Port \hookrightarrow_{fin} Filter_{I,L} \\
relationMapping^A_{I,L} & : & Relation \hookrightarrow_{fin} Dependency_{I,L} \\
roleMapping^A_{I,L} & : & Role \hookrightarrow_{fin} \mathbb{N}_0 \\
linkMapping^A_{I,L} & : & Link \hookrightarrow_{fin} Quantifier_{I,L}
\end{array}
$$

Note that these mapping functions are parameterized by the architecture A and implementation I of some software system under consideration (as well as by the programming language L). Also note that the mapping functions are partial functions. For example, consider the concept mapping. All concepts that do not belong to some architectural view of conceptual architecture A are not mapped to anything. The same holds for the port mapping, relation mapping, role mapping and link mapping. They are also finite because every architecture has a finite number of architectural views, each with a finite number of concepts, relations, ports, roles and links.

In order for an architectural instantiation to be well-formed, we need to specify some additional well-formedness constraints on the above mapping functions. For example, regarding the role mapping, we must ensure that for every architectural relation:

- every role of that relation is mapped to a number between 1 and the total numbers of roles belonging to that relation;
- every role is mapped to a unique number: no two relation roles can have the same number;
- there are no numbers left unassigned: every number has a role mapped to it.

All these constraints can be captured in a single bijectiveness constraint between the set of roles belonging to the relation and the set {1,...,n}, where n is the total number of roles belonging to the relation.

Well-formed role mapping.
Let L be some programming language, $I \in Implementation_L$ and $A \in Architecture$;

$\forall r \in Relations_A : roleMapping^A_{I,L\ |\ roles(r)} : roles(r) \rightarrow \{\ 1, \ldots,\ |roles(r)|\ \}$ *is bijective*

Regarding the relation mapping, we must ensure that every relation is mapped to a virtual dependency of which the arity is equal to the total number of roles belonging to the relation.

Well-formed relation mapping.
Let L be some programming language, $I \in Implementation_L$ and $A \in Architecture$;

$\forall r \in Relations_A : |roles(r)| = arity(relationMapping^A_{I,L}(r))$

Of course, we also require that every entity (i.e., concept, port, relation, role or link) in some architectural view is mapped to exactly one architectural abstraction. Because the mapping functions are functions, it is trivial that no more than one architectural abstraction can be associated with such an entity. However, because the mapping functions are partial, we do need to ensure that there every architectural entity in a conceptual architecture has an architectural abstraction associated with it.

Well-formed architectural instantiation.
Let L be some programming language, $I \in Implementation_L$ and $A \in Architecture$:

$conceptMapping^A_{I,L}(c)$	*is defined*	$\Leftrightarrow$	$c \in Concepts_A$
$relationMapping^A_{I,L}(r)$	*is defined*	$\Leftrightarrow$	$r \in Relations_A$
$linkMapping^A_{I,L}(l)$	*is defined*	$\Leftrightarrow$	$l \in Links_A$
$portMapping^A_{I,L}(p)$	*is defined*	$\Leftrightarrow$	$p \in Ports_A$
$roleMapping^A_{I,L}(r)$	*is defined*	$\Leftrightarrow$	$r \in Roles_A$

5.4.5 Formalizing architectural conformance checking

In essence, checking conformance of an implementation I to a conceptual architecture A involves not much more than the transformation of A into a logical expression. (As we will see, the constructed expression is not first-order but second-order, because it contains some second-order variables representing quantifiers.) The truth value of the constructed expression indicates whether the implementation I is conform to the conceptual architecture A. In this subsection, we formalize the construction of such a logical expression in terms of a semantic function $[\ ]_{I,L}$ on conceptual architectures. Note that the semantic function is parameterized with the implementation I under consideration, as well as with the programming language L.

Architectural conformance.
An implementation I in programming language L conforms to a conceptual architecture A
$\Leftrightarrow \quad [\ A\ ]_{I,L} = True$

We define this semantic function $[\ ]_{I,L}$ compositionally. More precisely, the semantics of conceptual architectures is defined in terms of the semantics of architectural views, which is in turn defined in terms of the semantics of architectural relations.

Because a conceptual architecture consists of multiple architectural views, we define the semantics of a conceptual architecture in terms of the semantics of its architectural views. More precisely, for some conceptual architecture A, $[\ A\ ]_{I,L}$ is the conjunction of all $[\ V\ ]^A_{I,L}$ for all

architectural views V belonging to A. This formalizes the intuition that an implementation is conform to some conceptual architecture A if and only if it is conform to each architectural view in A.

Semantics of conceptual architectures.
Let L be some programming language, $I \in Implementation_L$ and $A \in Architecture$;

$$[\ A\ ]_{I,L} = \bigwedge_{V \in views(A)} [\ V\ ]^A_{I,L}$$

Note that this definition implies that the semantics of an empty conceptual architecture (i.e., one with no architectural views) is always true. In other words, every implementation is conform to a conceptual architecture that does not contain any architectural view.

The semantics of an architectural view V can be defined in terms of the semantics of the architectural relations in that view. More precisely, for some architectural view V, $[\ V\ ]^A_{I,L}$ is the conjunction of all $[\ r\ ]^{A,V}_{I,L}$ for all architectural relations r in V. This formalizes the intuition that an implementation is conform to some architectural view if and only if it satisfies the constraints expressed by each of the architectural relations in that view.

Semantics of architectural views.
Let L be some programming language, $I \in Implementation_L$, $A \in Architecture$ and $V \in views(A)$;

$$[\ V\ ]^A_{I,L} = \bigwedge_{r \in relations(V)} [\ r\ ]^{A,V}_{I,L}$$

This definition implies that the semantics of an architectural view which contains no architectural relations is always true, regardless of whether or not it does contain architectural concepts. In other words, every implementation is in conformance with an architectural view without architectural relations. This is because in our architecture language, architectural relations are the only way to impose high-level constraints over implementation artifacts. If there are no architectural relations, the implementation does not need to satisfy any architectural constraint.

Finally, we need to define the semantic function on architectural relations. However, because this definition will be rather elaborate we first define an auxiliary function $\langle\ \ \rangle^A_{I,L}$ to compute the semantics of a concept or a port.[8] Intuitively, the semantics of an architectural concept c is the set of implementation artifacts that is described by the classification to which that concept is mapped.

Semantics of architectural concepts.
Let L be some programming language, $I \in Implementation_L$, $A \in Architecture$ and $c \in Concepts_A$;

$$\langle\ c\ \rangle^A_{I,L} = conceptMapping^A_{I,L}(c)$$

Note that it is not prohibited for the classification associated with c to be empty. In that case the semantics of c is the empty set. Also note that the concept mapping is undefined for concepts that do not belong to A.

The semantics of a port p of concept c is the set of all implementation artifacts belonging to that concept (or more precisely, to its semantics), filtered by some filter F to which the port p is mapped.

[8] We use a different notation for this semantic function because it does not return a Boolean, but a software classification.

Semantics of concept ports.
Let L be some programming language, $I \in Implementation_L$, $A \in Architecture$, $c \in Concepts_A$ and $p \in ports(c)$;

$$\langle\, p \,\rangle^A_{I,L} = F(\, \langle\, c \,\rangle^A_{I,L}\,) \qquad where \qquad F = portMapping^A_{I,L}(p)$$

Again, we note that the semantics of the port can be the empty set if the classification associated with the port's concept is empty (or if the filter associated with the port rejects all artifacts in the classification).

Now that we have defined these auxiliary semantic functions on architectural concepts and ports, we can finally define the semantics of architectural relations. We first give a formal definition. Then we intuitively explain the different parts of the definition.

Semantics of architectural relations.
Let L be some programming language, $I \in Implementation_L$, $A \in Architecture$, $V \in views(A)$ and $r \in relations(V)$;

Let $n = |roles(r)|$ *and* $F = curry_{I,n}(relationMapping^A_{I,L}(r))$;

$\forall\, i \in \{1, \ldots, n\}$, *let* $role_i \in roles(r)$ *such that* $roleMapping^A_{I,L}(role_i) = i$;

$\forall\, i \in \{1, \ldots, n\}$, *let* $Ports_i = \{\, p \mid (p, role_i) \in links(V) \,\}$;

$$[\, r \,]^{A,V}_{I,L} = \bigvee_{p_1 \in Ports_1} Q_{p_1}(\, C_{p_1},\ \bigvee_{p_2 \in Ports_2} Q_{p_2}(\, C_{p_2},\ \ldots,\ \bigvee_{p_n \in Ports_n} Q_{p_n}(\, C_{p_n},\ F\,)\ldots\,)\,)$$

where $C_{p_i} = \langle\, p_i \,\rangle^A_{I,L}$

and $Q_{p_i} = linkMapping^A_{I,L}((p_i, role_i))$

Now, let us take a closer look at this definition. We want to compute the semantics of some architectural relation r which has n roles. This relation r is mapped to a virtual dependency R. From the well-formedness constraint on relation mappings we know that R has arity n. We then transforming R into a curried function $F = \lambda x_n \ldots \lambda x_2.\lambda x_1.f(x_1, x_2, \ldots, x_n)$. The semantics of the relation r is the expression $f(x_1, \ldots, x_n)$ with the x_i's bound to the appropriate values.

So let us explain to which values each of the variables x_i will be bound. The purpose of the role mapping is to associate every role of r with exactly one of the arguments x_i of R. Let $role_i$ be the role that corresponds to argument x_i. From the well-formedness constraint on the role mappings, we know that this role is uniquely defined. If this $role_i$ is linked to a single port p_i (we will explain later what happens if $role_i$ is linked to multiple ports), we want to consider as set of values for x_i the filtered classification C_{p_i} to which p_i is mapped. Note that $\langle\, p_i \,\rangle^A_{I,L}$ is defined to compute this classification. So the semantics of r is the expression $f(x_1, \ldots, x_n)$, where $x_i \in C_{p_i}$ for each $i \in \{1, \ldots, n\}$.

But how do we apply the $x_i \in C_{p_i}$ over the expression $f(x_1, \ldots, x_n)$? This is determined by the quantifier Q_{p_i} to which the link between $role_i$ and port p_i is mapped. For example, if Q_{p_i} is the $Forall_{I,L}$ quantifier, we evaluate the expression $f(x_1, \ldots, x_n)$ for each $x_i \in C_{p_i}$. In other words, we take the conjunction of all expressions that are obtained by substituting x_i in $f(x_1, \ldots, x_n)$ one by one for each of the values in C_{p_i}. So the semantics of r is the expression

$$Q_{p_1}(C_{p_1}, Q_{p_2}(C_{p_2}, \ldots, Q_{p_n}(C_{p_1}, \lambda x_n \ldots \lambda x_2.\lambda x_1.f(x_1, x_2, \ldots, x_n)) \ldots))$$

where each Q_{p_i} is the quantifier associated with the link between port p_i and $role_i$ and where each $C_{p_i} = \langle\, p_i \,\rangle^A_{I,L}$.

The only thing we did not yet take into account is that $role_i$ may be linked to more than one port p_i. So we define $Ports_i$ as the set of all ports p_i to which $role_i$ is linked. Now, recall from Subsections 4.2.2 and 5.2 that when multiple ports are linked to the same role, this is interpreted as a disjunction. Therefore, for each $x_i \in C_{p_i}$ we need to take the the disjunction over all ports

$p_i \in Ports_i$, which finally yields the expression

$$\bigvee_{p_1 \in Ports_1} Q_{p_1}(\, C_{p_1},\ \bigvee_{p_2 \in Ports_2} Q_{p_2}(\, C_{p_2},\ \ldots,\ \bigvee_{p_n \in Ports_n} Q_{p_n}(\, C_{p_n},\ F\,)\ldots)\,)$$

As a concrete example of the semantic function on architectural relations, consider the architectural relation $r = \textit{Is Created By}$ on Figure 5.2 of page 61. It has a port named 'Created', which is connected to the architectural concept 'Auxiliary Application', and a port named 'Creator', which is connected to the architectural concepts 'User Application' and 'Auxiliary Application'. In other words, for this relation r, we have:

$n = 2$

$F = \lambda x_2.\lambda x_1.isCreatedBy_{C,C}(x_1, x_2)$

$role_1$ = 'UserInteraction.IsCreatedBy.Created'

$role_2$ = 'UserInteraction.IsCreatedBy.Creator'

$Ports_1$ = { 'UserInteraction.AuxiliaryApplication.Type' }

$Ports_2$ = {'UserInteraction.UserApplication.Type','UserInteraction.AuxiliaryApplication.Type'}

$Q_{UserInteraction.AuxiliaryApplication.Type_1} = Forall_{I,L}$

$Q_{UserInteraction.UserApplication.Type_2} = Exists_{I,L}$

$Q_{UserInteraction.AuxiliaryApplication.Type_2} = Exists_{I,L}$

Hence, the semantics of $r = \textit{Is Created By}$ is the expression

$$Forall_{I,L}(\, C_{Aux},\ (\, Exists_{I,L}(\, C_{User},\ F\,)\ \vee\ Exists_{I,L}(\, C_{Aux},\ F\,)\,)$$

where C_{User} is the set of artifacts in the virtual classification to which the 'User Application' concept is mapped and C_{Aux} is the set of artifacts in the virtual classification to which the 'Auxiliary Application' concept is mapped. By substituting the definitions for $Forall_{I,L}$, $Exists_{I,L}$ and F, this expression can further be refined to:

$$\bigwedge_{x_1 \in C_{Aux}} (\ \bigvee_{x_2 \in C_{User}} isCreatedBy_{C,C}(x_1, x_2)\ \vee \bigvee_{x_2 \in C_{Aux}} isCreatedBy_{C,C}(x_1, x_2)\,)$$

or equivalently:

$$\forall\, x_1 \in C_{Aux} : (\, \exists\, x_2 \in C_{User} : isCreatedBy_{C,C}(x_1, x_2) \vee \exists\, x_2 \in C_{Aux} : isCreatedBy_{C,C}(x_1, x_2)\,)$$

This concludes our explanation of the semantic function, which formalizes the conformance checking algorithm. A more elaborate example will be worked out in Section 6.3 where we explain the implementation of the algorithm.

5.4.6 Discussion

To conclude this section, we discuss some problems and shortcomings of the current formalization and explain how these problems could be resolved.

A first minor flaw is that we required the quantifiers to be declared as part of the architectural abstraction language instead of as part of the ADL. Because these quantifiers provide important information to an architect on how a diagram should be interpreted, it would be better to consider them as part of the ADL. In this way, the architect is not forced to consider both the ADL and the architectural mapping to know how to interpret the links in some architectural view. Fixing this flaw requires only some superficial changes to the formalism. The ADL should allow for links to be annotated with quantifier symbols. Note that we already did this in Figure 5.2, although it was not supported by the architectural formalism. The architectural instantiation language should

still maps links to quantifiers, with an additional constraint that the quantifier to which a link is mapped is the one specified by the quantifier symbol attached to the link. (For example, a link annotated with a $\exists$ symbol should be mapped to an `exists` quantifier predicate.)

Of course, once quantifiers have been made explicit in the ADL, so must be the role numbers, otherwise one cannot correctly interpret an architectural diagram due to order of the quantifiers. (For example, $\forall x \in A : \exists y \in B : r(x, y)$ is not the same as $\exists y \in B : \forall x \in A : r(x, y)$.) The role numbers are needed to know in which order to apply the different quantifiers. Supporting explicit role numbers requires only some small changes to the formalism: the ADL should allow the annotation of roles with these numbers, and the architectural instantiation has an extra constraint that roles are mapped to the numbers with which they are annotated.

We already mentioned earlier that the direction of links is unimportant in our formalism: in our formalization, all links are undirected. Nevertheless, we still put arrows on the links in our diagrams, as a guideline for an architect on how to read the diagrams. When the role numbers are made explicit in the diagrams, these arrows are not needed anymore. The role numbers by themselves provide sufficient information. (Recall that we adopted the convention to associate an incoming link with the role representing the subject of the relation and outgoing links with the other roles. With role numbers, we can adopt a similar convention by associating number 1 with the role representing the subject of the relation.)

When multiple ports are attached to the same role, we assume a disjunctive semantics rather than a conjunctive one. This was indeed the intended semantics in the architectural views we considered. In those cases were we would prefer a conjunctive semantics, we can always achieve the same effect by splitting one architectural relation with multiple links attached to some role, into multiple architectural relations, each with one link attached to that role. After all, we know that the semantics of multiple relations is the conjunction of the semantics of those relations. However, this work-around is sometimes a bit cumbrous and leads to diagrams with a lot of redundancy. Therefore, a cleaner solution would be to make the intended semantics of multiple roles explicit on the diagrams. For example, we could annotate the links attached to such a port with a special notation indicating whether to interpret them disjunctively or conjunctively (or a combination of both, or maybe even another interpretation such as an exclusive disjunction). Of course, this special notation should be taken into account by the conformance checking formalism. This can be done by defining the semantics of an architectural relation as

$$[\ r\ ]^{A,V}_{I,L} = \Psi^1{}_{p_1 \in Ports_1} Q_{p_1}(\ C_{p_1},\ \Psi^2{}_{p_2 \in Ports_2} Q_{p_2}(\ C_{p_2},\ \ldots,\ \Psi^n{}_{p_n \in Ports_n} Q_{p_n}(\ C_{p_n},\ F\ )\ldots)\ )$$

where Ψ^i is a Boolean operator which specifies how to interpret the different ports that are linked to the i^{th} role of relation r.

In addition to the above shortcomings, we mention some special characteristics of our architectural formalism that are worth noting.

What is the semantics when multiple links are attached to the same port of some concept? If the links are connected to the roles of a different relation, these links should be interpreted conjunctively, simply because the semantics of an architectural view is the conjunction of the semantics of all architectural relations in that view. If multiple links are connected to the roles of a single relation, this simply means that the same port plays different roles in that relation.

As stressed before, we repeat that the semantics of an architectural view is simply the conjunction of all architectural relations in that view. No more, no less. In particular, this implies that architectural concepts only play a secondary role. The architectural concepts by themselves impose no constraints on an implementation. Only when they participate in an architectural relation will their semantics be taken into account. We stress this fact because it goes against the usual expectations that a concept in a software architecture declares the required presence of a corresponding component in the implementation.

A final comment has to do with the semantics of a concept that is mapped to an empty virtual classification. All ports of this concept will return an empty set. If such a port is linked to some role with a $\forall$ quantifier, the semantics of the relation will be automatically true, due to the

semantics of $\forall$. This may seem a bit awkward for those who are accustomed to more traditional architectural formalisms with components and connectors instead of concepts and relations. In such a formalism, the usual expectation when a component X is linked to some connector Y is that this represents a constraint of the form "there exists a component X in the implementation, which is related in a Y fashion to ...". In our formalism, on the other hand, the presence of an architectural concept does not necessarily imply the presence of some implementation component. For example, if a virtual classification attached to some concept is empty, and all ports of that concept are linked through universal links, the implementation may not implement the concept at all (since the virtual classification contains no implementation artifacts), and still be considered conform to the architecture.

To resolve this problem, we could put an extra constraint on the architectural mapping to disallow empty virtual classifications. In this way, every architectural concept will correspond to at least some implementation artifacts. But this still does not solve the problem entirely. We should also disallow concept ports to return empty classifications (i.e., they are not allowed to reject all artifacts in the classification associated with the port's concept). If not, we still have the same problem as above. Furthermore, it seems strange to link an empty port to some relation. Such a situation is more than likely an indication of some kind of problem with the architectural mapping.

5.5 Summary

Our architecture language defines the architecture of a software implementation in two parts. In the ADL, we describe what the conceptual architecture looks like. It consists of different architectural views which are each built up from architectural concepts, relations, ports, roles and links. In the AML, we declare the meaning of each of these architectural entities by mapping them to implementation artifacts and their dependencies. The AML provides a fixed set of architectural abstractions (i.e., virtual classifications, virtual dependencies, filters, argument numbers and quantifiers) which the different architectural entities are instantiated with, as well as a library of predefined predicates at different levels of abstraction (the DFW) in which terms these architectural abstractions are mapped to the implementation. The different kinds of architectural abstractions and the library predicates are independent of the particular software system under consideration, but many of the library predicates are Smalltalk-dependent.

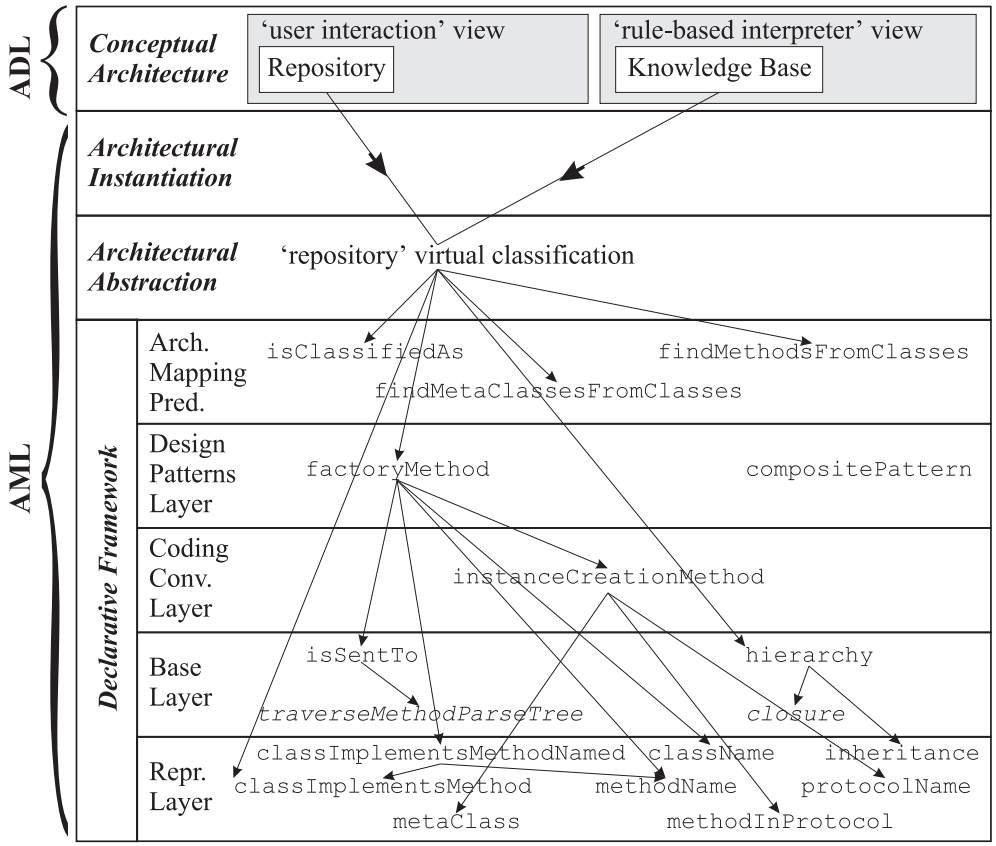

Figure 5.3: Overview of the declarative framework.

Figure 5.3 gives an overview of the different layers of the DFW, illustrates the dependencies between predicates in the different layers, and shows how the AML maps these predicates to architectural entities defined in the ADL.

In Section 5.4, we formalized the ADL and AML, as well as the conformance checking algorithm, in terms of a denotational semantics. More precisely, an implementation I defined in some programming language L is conform to a conceptual architecture A defined in the ADL, if the denotational semantics $[\ A\ ]_{I,L}$ is 'true'. The semantic function $[\ \ ]_{I,L}$ uses the architectural abstractions and architectural instantiation declared in the AML to relate the architecture A to the implementation I. In the next chapter (Section 6.3), we will sketch a Prolog implementation of the conformance checking algorithm.

Chapter 6

Implementing the Architecture Formalism using LMP

Now that we have defined the architectural formalism we show how it can be implemented straightforwardly in a logic meta-programming language. We sketch the setup of the logic meta-programming environment, implement the architecture language (i.e., the architecture description language, architectural instantiation language, architectural abstraction language and the declarative framework) and show the implementation of the conformance checking algorithm. We conclude the chapter with some future extensions of the formalism and explain how these could also be implemented in the logic meta-programming language.

6.1 The logic meta-programming language

Before explaining how to implement the architectural formalism in a LMP language, we give an overview of the setup of the LMP environment in which we conducted our experiments.

6.1.1 Setup

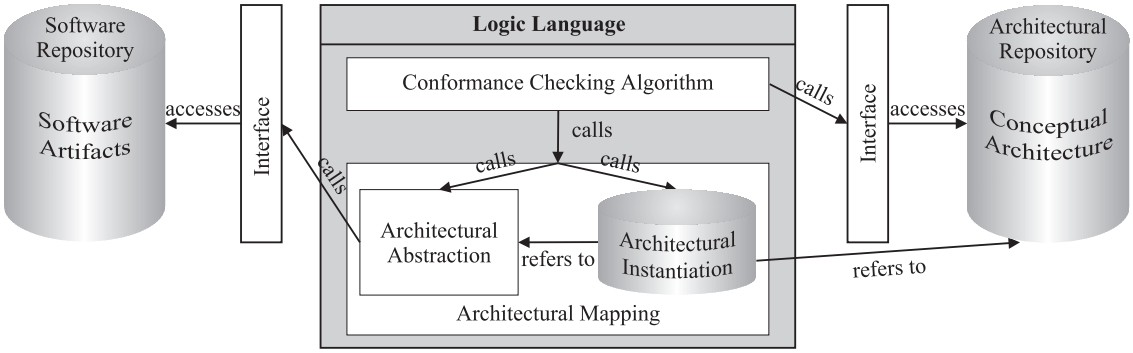

Figure 6.1: Schematic overview of the logic meta-programming setup.

Figure 6.1 gives a schematic overview of the setup of our LMP environment. To present this setup, we could have used the uniform architectural notation we have been using throughout this dissertation. However, to make the figure more understandable, we chose the following more specific notation:

- (logic) code fragments are represented as white rectangles with a thin border;
- data is denoted by cylinders;
- interfaces between applications are represented as white rectangles with a thick border;
- languages are rendered as shaded, labeled rectangles that may contain code and data;
- control flow and other relationships between the different elements are represented as labeled arrows.

This notation can be seen as a customization of our more general architectural notation. Section 8.3.4 elaborates on the need for customizable graphical representations of software architectures.

Since we use a LMP approach to check conformance of the implementation of a software system to its conceptual architecture, there are three main elements in the setup. First of all, we have a **Logic Language** in which both the *Conformance Checking Algorithm* and the *Architectural Mapping* are implemented. As these logic programs reason about the implementation artifacts of some software implementation, they are actually meta programs. The base-level implementation artifacts are stored in some **Software Repository** that can be accessed from within the logic language. The conformance checking algorithm verifies whether these artifacts satisfy the declared high-level architectural relationships. The architecture descriptions which describe the architectural structure to which the implementation artifacts should conform, are stored in some **Architectural Repository** which contains the conceptual architecture (i.e., the set of all architectural views).

As explained in Chapter 5, the architectural mapping consists of an *Architectural Instantiation* and an *Architectural Abstraction*. The architectural instantiation maps architectural entities to elements of the architectural abstraction. The architectural abstraction is an abstraction layer between the logic language and the software repository. The conformance checking algorithm uses the architectural instantiation to check whether the architectural abstractions conform to the constraints imposed by the conceptual architecture. Depending on how and where the implementation repository and architectural repository are represented, an extra **Interface** between the logic language (i.e., the architectural abstraction and the conformance checking algorithm, respectively) and these repositories may be needed.

The schematic overview of Figure 6.1 was deliberately kept as general as possible:

- It is left open which particular logic language is used. We experimented with two different logic languages: SOUL and Prolog.
- We did not specify how the software repository and architectural repository were represented. Although the figure seems to indicate that the repositories are external to the logic language, this does not have to be the case. For example, one or both repositories may be represented implicitly as a set of facts in the knowledge base of the logic language.
- Also, although the figure may suggest this, both repositories do not necessarily need to be physically distinct.
- Although our case study was mainly concerned with checking architectural conformance of a software implementation, we will see in Subsection 8.4.2 that the approach is equally suited for checking architectural conformance of other software artifacts, such as design models.
- We did not specify precisely what the optional interfaces between the logic language and the repositories look like. In fact, this strongly depends on how the repositories are represented. For example, this interface may be an ODBC interface (e.g., if the repository is an external database or tool repository), a translation layer, a combination of both, or may be left out entirely (e.g., if we use the internal fact base of the logic language as repository).

In the next subsections we discuss two specific instances of this general setup.

Setup of SOUL experiments

To validate our approach, we performed a case study using two alternative setups. In the first setup, the logic language SOUL was used to allow powerful logic reasoning about Smalltalk code fragments. This experiment was performed entirely in a Smalltalk development environment, as depicted in Figure 6.2.

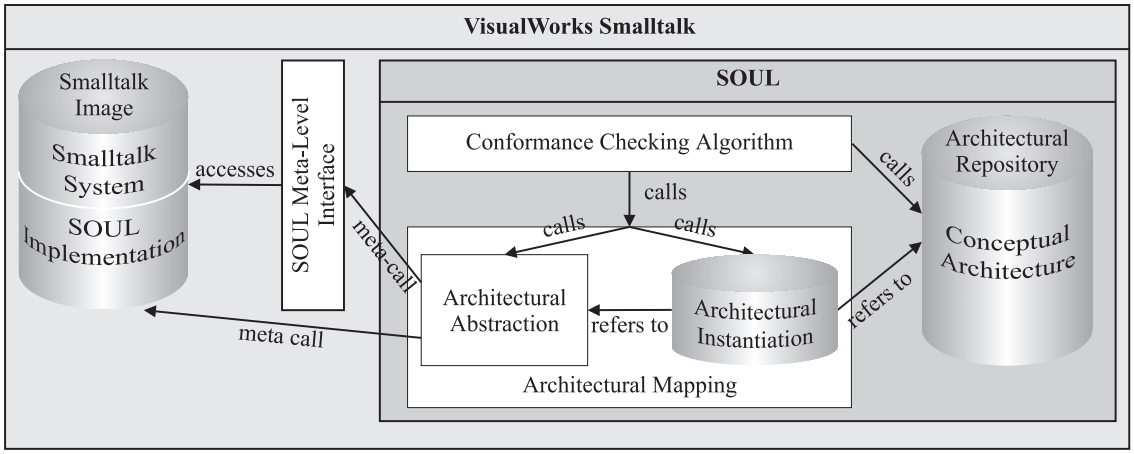

Figure 6.2: Setup for conformance checking in SOUL.

VisualWorks^{TM} Smalltalk was used as an implementation medium for SOUL. The SOUL language contains primitive constructs for evaluating Smalltalk code blocks inside logic rules. This allows SOUL expressions to reason about Smalltalk source code by making direct meta calls to the Smalltalk image. Alternatively, the image may be accessed through a meta-level interface for SOUL containing a predefined set of typical operations on the image. All architectural descriptions are stored directly as facts in the knowledge base of the SOUL language.

Setup of PROLOG experiments

In a second setup, standard Prolog was used as the logic meta language, and the implementation artifacts were stored in an external *Microsoft Access*^{TM} database. Accessing the database from within Prolog was done using an ODBC interface, called *Prodata*^{TM}. To translate the database tables to a more suitable representation, an additional repository-access layer was implemented in Prolog. As in the previous setup, the architectural descriptions were simply stored in the fact base of the logic language. This setup is sketched in the upper half of Figure 6.3.

The bottom half of the figure explains how the implementation repository was generated. The database containing the implementation artifacts is filled up front by means of a SOUL program. The reason we used SOUL for this purpose was precisely because of its powerful features for manipulating Smalltalk code. For doing the actual insertion of data values in the database, the SOUL program calls some Smalltalk ODBC primitives.

6.1.2 Logic language

SOUL

As mentioned above, we have experimented with two different, yet similar, logic languages: SOUL and Prolog. SOUL, being implemented entirely in Smalltalk, provided direct access to the Smalltalk image. It was more efficient in the sense that no slow connections to an external repository were needed and that efficient predefined Smalltalk methods for browsing the Smalltalk image could be called directly. Furthermore, the SOUL system was developed 'in house' (at the Programming Technology Lab) which allowed us to modify, extend or optimize it whenever needed.

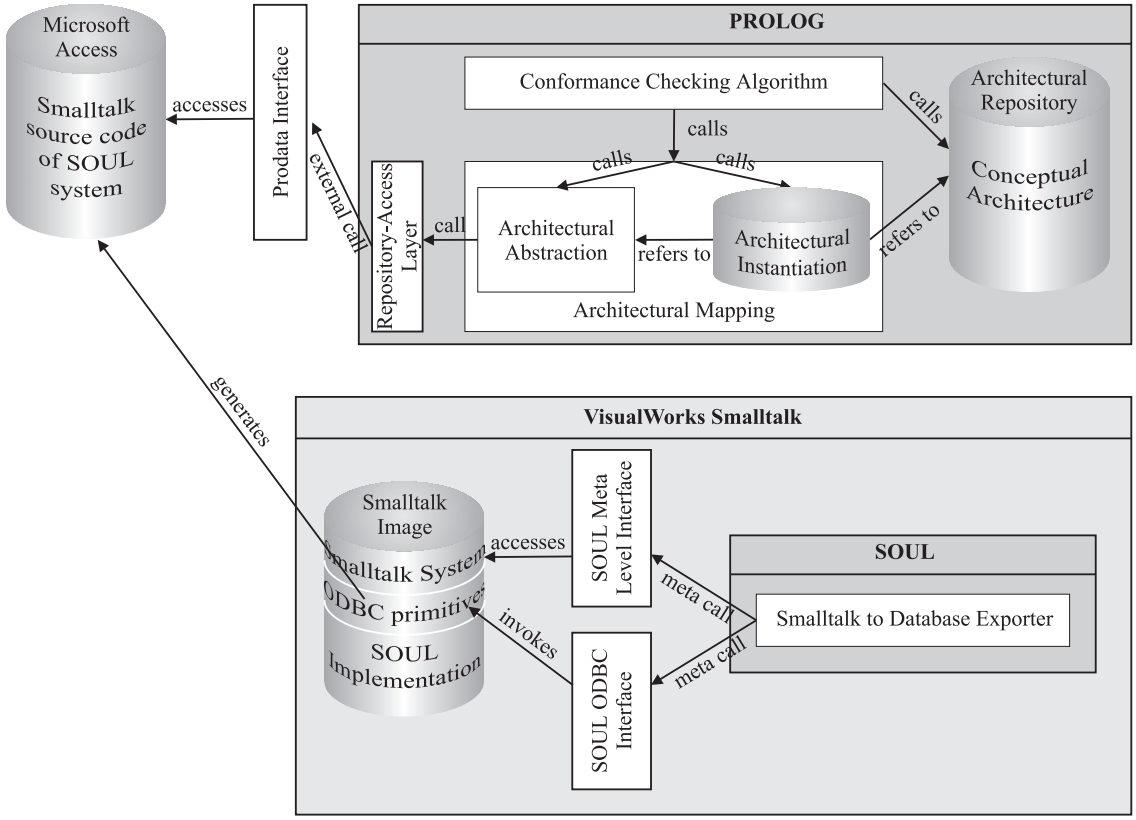

Figure 6.3: Setup for conformance checking in PROLOG.

Unfortunately, these advantages were also its main disadvantages. Because the SOUL language is strongly biased towards reasoning about Smalltalk code, it is not trivial to extend it to other kinds of software artifacts (e.g., other programming languages or design languages). Also, because SOUL was under constant evolution, existing and working SOUL programs often had to be re-implemented. A third problem was that because of the experimental nature of the SOUL language, its interpretation engine was less optimized than that of commercially available logic languages. It was also less easy to use or debug. (Debugging SOUL code typically had to be done by tracing the Smalltalk code of the interpreter.) For all these reasons — although the first experimental results with architectural conformance checking in SOUL were promising (see [52]) — we switched to a standard Prolog implementation for our later experiments.

PROLOG

Standard Prolog was chosen because it is well known and well documented, and because many efficient and well-supported commercial implementations are available on many platforms. In particular, we chose LPA *WIN-PROLOG*^{TM} because it had an optional *Prodata*^{TM} interface, which provides a tight coupling between the Prolog language and all ODBC 2 compliant database management systems [42]. In this way, we had all the advantages of using standard Prolog, while still being able to reason about data (e.g., implementation artifacts) stored in a database or any other ODBC-compliant repository.

6.1.3 Implementation repository

Smalltalk image

In the SOUL setup, the implementation repository is simply the Smalltalk image. It can be accessed either directly or through a special meta-level interface. This meta-level interface is a predefined set of Smalltalk expressions, encapsulated in Smalltalk methods, which are often used inside SOUL rules to manipulate the Smalltalk image. Instead of having to call these expressions explicitly from within SOUL, a simple meta call to one of the methods suffices to access the Smalltalk image. For example, when using SOUL to reason about Smalltalk implementations, we often want to compute the inheritance hierarchy of some class. Therefore, the meta-level interface has a predefined method `hierarchy` which invokes some specific methods in the Smalltalk system to compute this hierarchy. Apart from reducing duplication of complex Smalltalk expressions inside logic rules, this meta-level interface makes the SOUL code more independent of the particular Smalltalk system used (only the meta-level interface needs to be updated). Usage of the meta-level interface is not enforced by SOUL, however. It still remains possible to evaluate arbitrary Smalltalk expressions within logic rules.

External database

In the Prolog setup, the most straightforward choice for representing the implementation repository would be to store all implementation artifacts as facts in Prolog memory. However, due to the vast amounts of data needed, this was infeasible. Instead, we opted to store all artifacts in an external database that could be accessed through the *Prodata*^{TM} ODBC interface, which allows database tables to be accessed transparently[1] from Prolog as though they existed within the Prolog environment as facts.

Using an external repository that is accessed through ODBC has the advantage that we can choose any ODBC-compliant database management system. For example, to reason about another software implementation, we merely have to provide another database in which the implementation artifacts for that implementation are stored. Of course, both databases should have the same database scheme, so that the Prolog predicates that transparently access the database through the *Prodata*^{TM} interface remain valid. When this database scheme is carefully designed, it may even be possible, with a minimal effort, to reason about other kinds of software artifacts. For example, we want to use the same database scheme for representing either Smalltalk source code, Java source code or even UML class diagrams[2].

Of course, using an external database has several disadvantages as well. First of all, accessing the database externally through ODBC is less efficient than when the data would be stored directly in Prolog memory or in an internal repository such as in the SOUL case. To improve the efficiency, predefined and highly optimized SQL queries may be defined, but this comes at the cost of loosing generality, reusability and portability. Secondly, there is the overhead involved in exporting all implementation artifacts to the database. Related to this is the problem that the database should be updated every time the implementation is modified. Of course, we may be lucky to work in a development environment or CASE tool of which the repository can be accessed directly through ODBC. In that case, there is no generation overhead, and the repository is always up to date. However, because the repository probably uses a database scheme that differs from the one we expect, we may need to add an extra translation layer to access that repository. This translation layer may either be implemented in Prolog, or it may be part of the repository (e.g., under the form

[1] *Prodata*^{TM} facilitates the use of Prolog rules over the contents of the database, with no need to download any part of the database, as all database accesses are done on the fly. Backtracking, cut, call, negation and all other standard Prolog mechanisms work identically over the table accesses and the internal database, thus achieving the highest level of transparency possible [42].

[2] In the context of an industrial research project [51], experiments have been carried out in which UML class diagrams were extracted from some CASE tool and stored in the same database format we use for storing Smalltalk implementation artifacts. Even more, the same primitive Prolog predicates were used to manipulate and to reason about the data in this database.

of virtual tables). Again, this translation may have a negative impact on the efficiency. To avoid the extra translation layer we could actually re-implement all Prolog predicates that access the repository, but this is not only a lot more work, the Prolog code would also become less portable.

In our Prolog setup, we were not able to access the *VisualWorks*^{TM} Smalltalk repository directly through ODBC. (Although some Smalltalk ODBC primitives were available, they could only be used for export purposes, not for accessing the Smalltalk image.) Therefore, as explained, we had to write a generator to export all relevant implementation artifacts in the Smalltalk image (or part of it) to a database in the correct format. In spite of some technical problems caused by the collaboration of many different tools (*VisualWorks*^{TM}, SOUL, ODBC and *Microsoft Access*^{TM}), this was a fairly simple task, mainly thanks to the powerful abilities of SOUL to reason about object-oriented code at a high level of abstraction, and thanks to the simplicity of ODBC. The generator uses a set of ODBC primitives, implemented in Smalltalk, that allow a Smalltalk system to access an external database. After having made available these primitives to the SOUL language, which is also implemented in Smalltalk, we merely had to write a fairly straightforward SOUL program to extract each relevant artifact from the Smalltalk repository, and export it to the database. The result was a database in the desired format, containing all implementation artifacts of part of the Smalltalk system (more precisely, we only exported all Smalltalk categories and classes that implemented the SOUL system).

6.1.4 Architectural repository

Both in the SOUL and in the Prolog setup, we stored the architecture descriptions directly in the fact base of the logic language. Since there are typically only a few of those architecture descriptions (as compared to the huge amount of implementation artifacts), they can easily be stored in memory. In this way the descriptions can be retrieved efficiently, and we do not have to implement yet another repository and interface between the logic language and this repository. If, for some reason, we would prefer to store the architectural descriptions in an external repository, an approach similar to that of the previous subsection could be followed.

6.1.5 SOUL versus PROLOG

As a final remark we want to stress that we deliberately tried to keep both alternative setups as similar as possible. For example, because SOUL's syntax is similar to Prolog's, it is easy to automatically translate Prolog facts and rules into SOUL and vice versa. When switching from SOUL to Prolog, we actually extended the SOUL system to export all SOUL code in Prolog format. More recent versions of the SOUL system include an option to use Prolog syntax instead of SOUL syntax and to switch between both notations. All logic code fragments in this dissertation are shown in Prolog-syntax.

We also tried to keep the primitive Prolog predicates that transparently access an external *Microsoft Access*^{TM} database, as similar as possible to the primitive SOUL predicates that access the internal Smalltalk repository. For this purpose, we implemented an additional repository-access layer in Prolog which hides some of the dirty details of the *Prodata*^{TM} interface as well as the database format. (The computational overhead of this extra abstraction layer was minimal.) In this way, many of the higher-level predicates are exactly the same (up to a change in syntax) for both approaches. This makes it fairly easy to return to SOUL for future experiments, which still has the advantage of providing a single environment in which the code repository is always up to date. At the moment of writing this dissertation, however, even with the overhead of having to access an external repository, the Prolog setup still turned out to be more efficient than the SOUL setup.

6.2 Implementing the architecture language

Now that we have sketched the setup of the LMP environment, in this section we explain how to represent the architecture language in that environment. The architecture language consists of three different languages (the architecture description language, the architectural instantiation language and the architectural abstraction language) plus a layered declarative framework which serves as a library of predefined architectural mappings. As we adopt a LMP approach, we defined each of these languages and the DFW on top of our general LMP language. We refrained from inventing a special-purpose syntax for each of these languages (although it would be no problem whatsoever to do so). As a consequence, the models expressed in each of these languages are nothing more than declarations in the underlying LMP language. Of course, not every logic meta program is a well-formed program in one of these languages. Every language imposes its own specific constraints on the expected format of the declarations. In this section, we explain what the declarations look like for each of the languages. We also show the implementation of some of the predefined mapping predicates of the DFW.

6.2.1 Implementing the architecture description language

The ADL is very simple. A conceptual architecture described in this ADL is represented as a set of logic facts. There is one fact for every architectural view, which has the following format:

```
view(soul, soulUserInteraction).
```

The first argument represents the name of the conceptual architecture, and the second argument is the name of an architectural view in that conceptual architecture.

There is also one fact for every entity in an architectural view. All these facts have essentially the same format. The name (label) of the fact is the kind of entity it represents and the first argument is the name of the architectural view to which this entity belongs. For concepts and relations we additionally mention their name (which is unique in the architectural view). For example, the **User Application** concept in the 'user interaction' view is represented by the fact:

```
concept(soulUserInteraction, userApplication).
```

For concept ports (respectively, relation roles), we mention their name as well as the name of the concept (respectively, relation) to which they belong. For example, the following fact declares that the **User Application** concept has a port named Type.

```
port(soulUserInteraction, userApplication, type).
```

Links are declared by indicating the port and role that are linked. Because ports have a name that is unique only in the concept to which it belongs, to identify the port, we need to specify both the concept's name and the port's name. The same holds for roles. For example, the link between the Type port of **User Application** and the Interrogator role of the $Asks_1$ relation is described by the fact:

```
link(soulUserInteraction, userApplication, type, asks1, interrogator).
```

A concrete example of an architectural view described in the above format will be given in Subsection 7.1.1. Of course, this way of describing or inspecting architectural views is rather verbose. Therefore, it would be useful to have a tool which allows architects to input their architectures graphically (e.g., using the notation of Figure 4.1) and automatically convert this graphical representation to logic facts such as the above. (See Subsection 8.3.4.)

6.2.2 Implementing the architectural abstraction language

The architectural entities declared in the ADL are mapped to constructs of the architectural abstraction language. These constructs represent high-level abstractions of implementation artifacts and their dependencies. Each of the different kinds of constructs (i.e., virtual classifications, virtual dependencies, etc.) are defined in terms of the predefined mapping predicates provided by the DFW. In this subsection, we discuss and illustrate the format of these constructs.

Virtual classifications

One of the most important constructs of the architectural abstraction language is the virtual classification. It defines a set of implementation artifacts intentionally, by means of some high-level logic predicate. Every virtual classification is codified by a set of logic rules of the form

```
classifiedAs(ArtifactKind(VCName), Artifact) :-
     some logic code that declares when a certain Artifact of kind
     ArtifactKind belongs to the classification named VCName
```

In the above template code, the parameter *VCName* should be filled in with the name of the virtual classification being defined and the parameter *ArtifactKind* with the kind of artifact (e.g., class, method, instance variable) for which we are defining the virtual classification. There should be a rule for every kind of artifact of which there are elements in the classification. If a certain kind of artifacts, say instance variables, is not needed in the classification, no separate rule should be defined for that *ArtifactKind*. The logic code in the body of the rule can make use of any of the predicates provided by the DFW. However, one should always try to use predicates defined in the highest layers of the framework and try to avoid as much as possible to use low-level predicates. Also, if no high-level predicate is available for your purpose, it is better to define one first, add it to the framework, and then use it rather than hard-coding the implementation of this new high-level predicate in the body of the virtual classification.

As an illustration, consider the following definition of the virtual classification 'userApplication'. It consists of two rules, one defining which classes belong to the classification (based on their category and name) and one defining which methods belong to it (based on the classes that were already classified). There are no other rules as the classification contains only methods and classes but no other kinds of artifacts. The details of this example will be explained in Subsection 7.1.3.

```
classifiedAs(class('userApplication'), Class) :-
    categoryName(Category, 'SOULUIApplications'),
    classInCategory(Category, Class),
    className(Class, ClassName),
    patternMatch(ClassName, and(prefix('SOUL'), postfix('App'))).

classifiedAs(method('userApplication'), Method) :-
    findMethodsFromClasses(Method, 'userApplication').
```

Filters

Filters are represented as unary logic predicates that take an `Artifact` as input and check whether or not this Artifact is accepted by the filter. In other words, they have the following format:

```
FilterName(Artifact) :-
     some logic code that succeeds if the Artifact should be accepted
     or fails if the Artifact is to be rejected
```

The name of the filter, *FilterName*, should be chosen so that it reflects the purpose of the filter. For example, a filter that only accepts methods and rejects anything else is named `methodFilter`; a filter that accepts anything is named `identityFilter`.

In general we want to keep the filters as simple as possible. In practice, we use the filters only to select the artifacts of a certain kind from a classification, and leave the computation of these artifacts to the virtual classification.

Virtual dependencies

A virtual dependency is a declaratively defined relationship among implementation artifacts:

```
VirtualDependencyName(Artifact1, ..., ArtifactN) :-
     check whether some n-ary relationship holds among
     implementation artifacts Artifact1 to ArtifactN
```

An n-ary architectural relation will be mapped to an n-ary virtual dependency predicate. Its roles will be mapped to the arguments of this predicate. Note that, if necessary, the definition of the predicate may consist of several rules (all with the same head *VirtualDependencyName* and the same number of arguments).

The name of the virtual dependency, *VirtualDependencyName*, should be chosen so that it gives an indication of the relation to be checked. Furthermore, we adopted the convention to end every *VirtualDependencyName* with an indication of the expected argument types (`M` stands for method, `C` for class, etc.). For example, `mentions_M_M` is a virtual dependency which checks whether there is a 'mentions' relationship between two methods.

Argument numbers

The roles of an architectural relation are mapped to integers ranging from 1 to the total number of roles of the architectural relation. Every number in this range is associated with exactly one role of the relation.

Quantifiers

We represent a `Quantifier` by a second-order logic predicate of the form

```
Quantifier(Generator, Test)
```

The idea is that the `Generator` predicate generates possible values to which the `Test` predicate should be applied. (The `Generator` will correspond to a virtual classification and the `Test` to a virtual dependency.) To which values and how exactly the application is performed depends on the predicate. A typical example of such a *Quantifier* is the primitive second-order logic predicate `forall`. `forall(Generator, Test)` checks, for all solutions of `Generator`, whether `Test` is true. In other words, the `Test` predicate is applied to each of the generated values, in the order in which they were generated. The application should succeed for *all* values.

6.2.3 Implementing the architectural instantiation language

Just like every architectural view described in our ADL is represented as a set of logic facts, its architectural instantiation will also be represented as a set of logic facts. There is one fact for each architectural entity, declaring the architectural abstraction to which that entity is mapped.

Concept mappings associate virtual classifications with architectural concepts in some architectural view and are declared by facts of the form:

```
conceptMapping(ArchitecturalView, Concept, VCName).
```

were *VCName* is the name of the virtual classification.

A port mapping associates a port of an architectural concept with a filter that acts on the virtual classification associated with the concept and is defined by a fact of the form:

```
portMapping(ArchitecturalView, Concept, Port, Filter).
```

Relation mappings are facts of the form:

```
relationMapping(ArchitecturalView, ArchitecturalRelation, VirtualDependency).
```

They map an architectural relation in some architectural view to some virtual dependency. For this mapping to be correct, the number of arguments of the virtual dependency must be equal to the number of roles of the relation to which it is associated.

Role mappings map a role of some architectural relation to an argument (number) of the virtual dependency associated with the relation. Hence, role mappings are declared by facts of the form:

`roleMapping(`*`ArchitecturalView, Relation, Role, ArgumentNumber`*`).`

Finally, a link mapping maps a link to its associated quantifier, by means of a fact of the following form:

`linkMapping(`*`ArchitecturalView, Concept, Port, Relation, Role, Quantifier`*`).`

For a concrete example of an architectural instantiation in the above format we refer to Subsection 7.1.2.

6.2.4 Implementing the declarative framework

Finally, we take a look at how the DFW is implemented in our LMP language. We show the implementation of some of the predefined predicates in each layer of the DFW. Only the sublayers of the logic meta-programming layer (i.e., the logic layer and the repository-access layer) are not discussed because of their very technical nature, and because most predicates in these layers are hard-coded primitives of the LMP language.

Representational layer

Internally, in our LMP language, we represent implementation artifacts as data structures of the form `ArtifactKind(ArtifactName, Identifier)`.[3] `ArtifactName` is the name of the artifact and `ArtifactKind` is its kind (e.g., class, method or instance variable). The `Identifier` is a unique number that is used internally when other information associated with the artifact needs to be retrieved from the repository. Whenever we are not interested in this identifier, it will be written as a variable. Some examples:

```
class('SOULTerms',1989)
metaclass('SOULTerms class',_)
method('at:',1992)
method('interpret:repository:',1279)
method('interpret:repository:',1651)
argument('term',735)
methodProtocol('instance creation', 73)
methodProtocol('instance creation', _)
...
```

It is possible to have multiple artifacts with the same kind and name, but with a different identifier. For example, there may be multiple methods named 'interpret:repository:' that belong to different classes. They can only be distinguished by their identifier, or by calling one of the primitive predicates which use the identifier to look up information in the repository. For example, the logic query `classImplementsMethod(C,method('interpret:repository:',1651))` has the unique result `C = class('SOULRule',1624)`.

The repository-access layer contains primitive predicates like `artifact(ID,Kind,Artifact)` to retrieve an `Artifact` with identifier `ID` of a certain `Kind` from the implementation repository, and `artifactNestingID(PartID,WholeID)` to check whether an artifact with identifier `PartID` is in a

[3] As the method parse-tree structure is often too verbose, methods will also be represented by this internal data structure instead of by their parse tree. In this way, we obtain a uniform data structure for all artifact kinds. However, the method parse trees are still cached in the repository, so that they can be retrieved whenever they are needed, based on a method's identifier.

nesting relationship with an artifact with identifier `WholeID`. The predicates of the representational layer are defined directly in terms of predicates such as these. Some representative examples are given below:

```
baseClass(Class) :-
    artifact(_ID, class, Class).

methodName(Method,MethodName) :-
    artifact(ID, method, Method),
    Method = method(MethodName, ID).

classImplementsMethod(Class,Method) :-
    artifact(MethodID,method,Method),
    ( artifact(ClassID,class,Class);
      artifact(ClassID,metaclass,Class) ),
    artifactNestingID(MethodID,ProtocolID),
    artifactNestingID(ProtocolID,ClassID).
```

Base layer

The base layer contains predicates for traversing method parse trees, type inferencing predicates and predicates that implement structural relationships.

As an example of the first kind of predicates, below we show the implementation of the `isSentTo` predicate. It is defined in terms of the generic predicate `traverseMethodParseTree`. The implementation of the other method parse-tree traversal predicates is similar. Other (including user-defined) predicates that need to traverse a method parse tree can also be defined in the same way.

```
isSentTo(ClassName, MethodName, Receiver, Message, Arguments) :-
    traverseMethodParseTree( ClassName, MethodName,
                             [Receiver, Message, Arguments],
                             foundMessageSend, processMessageSend).

% foundMessageSend specifies how to recognize a message send statement in the parse tree.
foundMessageSend(send(Receiver, Message, Arguments)).

% processMessageSend specifies how to retrieve the different parts of a message send.
processMessageSend(send(Receiver, Message, Arguments), [Receiver, Message, Arguments]).
```

Note that method parse-tree traversal predicates such as the one above require no knowledge of the representational layer, other than what the format is of method parse trees.

As a concrete example of a type inferencing predicate, we show the implementation of the predicate `instVarTypes(C,IV,Types)`, which computes a list of potential `Types` for an instance variable `IV` of some class `C`. The predicate yields an approximate answer only. It infers the possible types for the instance variable by statically looking at all the messages sent to that variable (from the class `C` up to the first superclass that implements the variable). All classes that understand all these messages are then accumulated in a type list `Types`. Every one of these classes is a possible candidate for the type of the variable (it is in general impossible to find the unique type statically). The precision of the answer depends on the amount of messages that are sent to the instance variable of which we need to compute the type. The more information we have, the more precise the type can be inferred (i.e., the less candidate types are generated).

```
instVarTypes(Class,InstVar,Types) :-
    % does Class or one of its superclasses contain a variable InstVar?
    instVarFlattened(Class,InstVar),
    % compute the set of all Messages sent to this variable
    instVarName(InstVar,InstVarName),
    instVarMessages(Class,InstVarName,Messages),
    Messages \= [], % Fail if no messages are sent to the variable
    % compute all classes that understand all these Messages
    findall(Type,understandsAll(Type,Messages),Types).
```

Another type-inferencing predicate is `returnType(Method, Type)` which infers the return `Type` of some `Method`. Note that it relies on two other base-layer predicates, namely `returnStatement` and `mayHaveType_E_M_C`.

```
returnType(Method, Type) :-
    classImplementsMethod(Class, MethodName, Method),
    className(Class, ClassName),
    returnStatement(ClassName, MethodName, Expression),
    mayHaveType_E_M_C(Expression, Method, Type).
```

As an example of a predicate implementing a structural relationship, we show the implementation of the `hierarchy` predicate, which is the transitive closure of the `inheritance` predicate.

```
hierarchy(Super,Sub) :-
    class(Super), class(Sub), closure(inheritance,Super,Sub).
```

Coding conventions layer

As a typical example of a method coding convention, we show the `abstractMethod` predicate. As explained in 5.3.5, abstract methods in Smalltalk can be recognized because they make a `subclassResponsibility` self send.

```
abstractMethod(Class,Method) :-
    findMethod(Class,Method,
               or( exact(`[send(variable(self),subclassResponsibility,[])]`),
                   exact(`[return(send(variable(self),subclassResponsibility,[]))]`))
               ).
```

The implementation of many other coding convention predicates (like `instanceCreationMethod`, `mutatorMethod` and `accessorMethod`) is given in Appendix B.

Design patterns layer

In Subsection 5.3.5, we mentioned two examples of predicates that describe the structure of some design pattern: `compositePattern` and `factoryMethod`. We do not include the exact implementation of the `compositePattern` predicate here; it is explained in detail in [86]. The `factoryMethod` predicate is implemented as follows:

```
% Does Method send an instance-creation message to Class?
factoryMethod(Class,Method) :-
    % Does Method send a message to Class?
    classImplementsMethodNamed(C,MN,Method), className(C,CN),
    isSentTo(CN,MN,Receiver,Message,Arguments),
    className(Class,Receiver),
    % Is the message an instance-creation message for Class?
    methodName(M,Message),
    instanceCreationMethod(Class,M).
```

Architectural mapping predicates

Finally, we show the implementation of some of the predefined architectural mapping predicates.

Virtual classifications. We distinguish four categories of architectural mapping predicates in terms of which to define virtual classifications:

1. predicates that were already defined in lower layers of the DFW;
2. predicates that compute virtual classifications from already defined ones;
3. predicates that implement operators on virtual classifications;
4. predicates that implement high-level dependencies among implementation artifacts.

We only discuss predicates of the second and third categories here. Predicates of the first category are implemented by the previous layers. Predicates of the fourth category will be explained in the paragraph on virtual dependencies.
All virtual classifications are defined by means of the `classifiedAs` predicate. Therefore, a first way to define a virtual classification in terms of an already existing one is by directly calling this predicate. Alternatively, we can use more high-level predicates like `findMethodsFromClasses`, `findMetaClassesFromClasses`, `findClassesFromMethods`, and so on. The implementations of these predicates are rather straightforward:

```
% Does Method belong to a class in the classification with name VCName?
findMethodsFromClasses(Method, VCName) :-
    classifiedAs(class(VCName), Class),
    classImplementsMethod(Class, Method).

% Is MetaClass the meta class of a class in the classification with name VCName?
findMetaClassesFromClasses(MetaClass, VCName) :-
    classifiedAs(class(VCName), Class),
    metaClass(Class, MetaClass).
```

Only the implementation of `findClassesFromMethods` is a bit more subtle: because a classification may contain multiple methods that belong to the same class, we need to remove duplicate classes.

```
% Does Class implement one of the methods in the classification with name VCName?
findClassesFromMethods(Class, VCName) :-
    findall( SomeClass,
             ( classifiedAs(method(VCName), Method),
               classImplementsMethod(SomeClass, Method) ),
             Classes),
    removeDuplicates(Classes, NoDups),
    member(Class, NoDups).
```

We can also define a virtual classification as a combination of other ones, using binary operators like `union`, `intersection`, `difference`, etc. Because virtual classifications are computed sets of implementation artifacts, these operators are defined in terms of the logic operators disjunction (`;`), conjunction (`,`) and negation (`not`).

```
union(VC1,VC2,Artifact) :-
    classifiedAs(VC1,Artifact); classifiedAs(VC2,Artifact).
intersection(VC1,VC2,Artifact) :-
    classifiedAs(VC1,Artifact), classifiedAs(VC2,Artifact).
difference(VC1,VC2,Artifact) :-
    classifiedAs(VC1,Artifact), not classifiedAs(VC2,Artifact).
```

Filters. There are two kinds of predefined filters. The 'trivial filters' always succeed or always fail. The 'kind filters' that only accept artifacts of a certain kind can be defined straightforwardly in terms of primitive predicates of the representational layer such as `baseClass`, `metaClass` and `method`.

```
% identityFilter is the trivial filter which accepts all artifacts
identityFilter(Artifact)  :- true.
% forgetfulFilter is the trivial filter which rejects all artifacts
forgetfulFilter(Artifact) :- fail.
% baseClassFilter is a filter which accepts only base (i.e., non-meta) classes
baseClassFilter(Artifact) :- baseClass(Artifact).
% metaClassFilter is a filter which accepts only meta classes
metaClassFilter(Artifact) :- metaClass(Artifact).
% methodFilter is a filter which accepts only methods
methodFilter(Artifact) :- method(Artifact).
...
```

Virtual dependencies. As concrete examples of virtual dependency predicates, we discuss two predicates that are needed for defining architectural relations in the 'user interaction' architectural view. The first is the predicate `mentions_M_M` which takes two methods as arguments and checks whether the one mentions the name of the other somewhere in its body. This is implemented in terms of the auxiliary predicate `findMethod`.

```
mentions_M_M(Method1,Method2) :-
    methodName(Method2,MN2),
    findMethod(_,Method1,contains(MN2)).
```

A second, more complex, example of a virtual dependency predicate is the `asks_C_M` predicate which checks whether (a method of) some class `C2` asks some method `M1` for information (and actually uses the returned information). It is defined in terms of a more primitive predicate `isAskedBy_M_M` which checks for an invocation relation using the `isSentTo` predicate. Next, it checks whether the found message send is actually used in the method performing the invocation, by means of the `isUsedBy_E_M` predicate. This auxiliary predicate uses a mixture of efficient string pattern matching and more precise (but less efficient) parse-tree traversing to check for all possible manners in which an expression may be used (assigning the value to a variable, passing the value as an argument to some method, returning the value, ...). We will revisit the issues of efficiency and precision in Subsection 7.1.7.

```
asks_C_M(C2,M1) :-
    classImplementsMethod(C2,M2),
    isAskedBy_M_M(M1,M2).

% isAskedBy_M_M(M1,M2) checks if M1 is asked for information by some method M2
isAskedBy_M_M(M1,M2) :-
    % First we check whether method M2 invokes method M1
    classImplementsMethodNamed(C2,M2Name,M2),
    className(C2,C2Name),
    methodName(M1,M1Name),
    isSentTo(C2Name,M2Name,Receiver,M1Name,Arguments),
    mayHaveType_E_M_C(Receiver,M2,ReceiverClass),
    classImplementsMethod(ReceiverClass,M1),
    % Then we check whether M2 actually uses the result of the message send
    isUsedBy_E_M(send(Receiver,M1Name,Arguments),M2).
```

Note that we also used the auxiliary predicate `mayHaveType_E_M_C` in the above predicate. This is because the `isSentTo` predicate does not check whether the method `M2` actually invokes `M1`; it only checks whether `M2` sends a message with the same *name* as the method `M1`. To be sure that

it is actually the method `M1` that is being invoked, we use the predicate `mayHaveType_E_M_C` to infer the type of the class that receives the message, and verify that this is indeed the class that implements `M1`.

For more examples of virtual dependency predicates we defer to Chapter 7.

Quantifiers. Two predefined quantifier predicates provided by the DFW are `forall` and `exists`, representing the set quantifiers $\forall$ and $\exists$. The second-order predicate `forall` is actually a primitive predicate of the logic layer. The quantifier `exists` can also be defined easily, by making use of the primitive second-order predicate `one` that is provided by the logic layer. `one(X)` checks for the first valid solution of some logic expression `X`. `exists(Generator, Test)` applies the `Test` predicate one by one to each of the generated values, in the order in which they were generated, until *one* is found for which `Test` succeeds.

```
exists(Generator, Test) :-
    one((Generator, Test)).
```

In addition to these two quantifier predicates, we also implemented some special versions of these predicates which report special information to the user in case of failure. Below, we show the implementation of two debugging versions of the `forall` predicate. In case of failure, `forallDebugOne` aborts on the first failure and prints the test that failed on screen. `forallDebugAll` does not abort after the first failure, but accumulates all subsequent failures as well and reports these to the user.

```
forallDebugOne(Generator, Test) :-
    one((Generator, not Test))  ->  ( write(Test), fail );
    otherwise                   ->  true.

forallDebugAll(Generator, Test) :-
    findall( Test,
             (Generator, not Test),
             Failures),
    ( Failures = [] -> true;
      otherwise     -> ( write(Failures), fail ) ).
```

At this time, no other quantifiers are implemented in the declarative framework, because we did not need any others for our case study. However, there is no problem whatsoever to add new ones to the framework if they would be needed.

6.3 Implementing the conformance checking algorithm

Based on the architectural formalism introduced in Chapter 5, we explain how the conformance checking algorithm is implemented in our LMP environment. We first discuss the conformance checking algorithm informally and then sketch its Prolog implementation.

6.3.1 Informal definition

When checking conformance of the implementation of a software system to a conceptual architecture, the conformance checking algorithm constructs a logic expression which corresponds to that conceptual architecture. The construction process of this logic expression is very similar to the denotational semantics we defined in Subsection 5.4.5. After having constructed the expression, conformance checking merely corresponds to computing its truth value. The expression contains logic variables, predicates representing relations, second-order predicates representing quantifiers and predicates representing (filtered) domains of values. Every relation predicate in the expression corresponds to an architectural relation, the logic variables correspond to the roles of that relation, the quantifier predicates correspond to the quantifiers associated with links, the predicates representing domains of values correspond to the virtual classifications associated with the architectural concepts, and the filter predicates correspond to the filters associated with the concept roles.

Now, let us illustrate in more detail how the logic expression is constructed, taking the architectural view of Figure 5.2 on page 61 as an example. For reasons of clarity, we explain the construction process as a sequence of transformations that transform an architectural view step by step, until the desired logic expression is obtained. Note that the actual conformance checking algorithm of Subsection 6.3.2 will perform all these transformations in one single pass.

Step 1

In a first transformation step, we transform the architectural view into a semi-formal formula, which is a conjunction of logic clauses, one clause for each architectural relation. For example, the architectural view depicted in Figure 5.2 is transformed into the following formula:

$$
\begin{aligned}
&\exists\, Trigger \in Event(InputWindow) : \\
&\qquad (\ \exists\, Action \in Request(UserApplication) : Activates(Trigger, Action) \\
&\qquad\ \vee \exists\, Action \in Request(AuxiliaryApplication) : Activates(Trigger, Action)\ ) \\
\wedge\ &\forall\, Created \in Type(AuxiliaryApplication) : \\
&\qquad (\ \exists\, Creator \in Type(UserApplication) : IsCreatedBy(Created, Creator) \\
&\qquad\ \vee \exists\, Creator \in Type(AuxiliaryApplication) : IsCreatedBy(Created, Creator)\ ) \\
\wedge\ &\forall\, Interrogator \in Type(UserApplication) : \\
&\qquad \exists\, Interrogated \in Interpret(QueryInterpreter) : Asks1(Interrogator, Interrogated) \\
\wedge\ &\forall\, Interrogator \in Type(UserApplication) : \\
&\qquad \exists\, Interrogated \in Request(Repository) : Asks2(Interrogator, Interrogated) \\
\wedge\ &\forall\, Creator \in Type(UserApplication) : \exists\, Argument \in Type(QueryResult) : \\
&\qquad \exists\, Created \in Creation(OutputViewer) : CreatesWith(Creator, Argument, Created).
\end{aligned}
$$

Because this transformation is fairly straightforward, we do not discuss it in detail here. We only mention that, as explained earlier, for an architectural relation with more than one link attached to one of its roles, a disjunction is used to enumerate all architectural concepts linked to this role. (For example, there is a disjunction for the 'Activates' relation to capture the fact that it interacts with either a 'User Application' or an 'Auxiliary Application'.)

In the previous formula, the order of the different conjuncts (and disjuncts) is unimportant. The order of the different subexpressions in one such conjunct is important, however. For example, as in mathematical logic, $\forall x \in A : \exists y \in B : r(x, y)$ does not mean the same as $\exists y \in B : \forall x \in A : r(x, y)$. We need to respect the intended order of the different quantifiers in each subexpression. This is precisely why we associated argument numbers with roles: the quantifiers attached to roles with a

lower argument number will precede those of roles with a higher number. The argument numbers also determine the order of the arguments occurring in the relations.

Step 2

The second transformation involves the replacement of:

1. the relation names with the names of the virtual dependency predicates to which they are mapped by the architectural instantiation (e.g., `Activates` is mapped to `mentions_M_M`)

2. the role names with logic variables

3. the conjunction symbol $\wedge$ by its equivalent Prolog operator ',' and the disjunction symbol $\vee$ by its equivalent Prolog operator ';'

4. the quantifier symbols $\forall$ and $\exists$ by the second-order logic predicates `forall` and `exists` that implement these quantifiers.

Performing all these replacements yields the following expression in Prolog pseudo-code:

```
exists( X1 IN Event(InputWindow),
        ( exists( X2 IN Request(UserApplication),
                  mentions_M_M(X1,X2) );
          exists( X2 IN Request(AuxiliaryApplication),
                  mentions_M_M(X1,X2) ) ),
forall( X3 IN Type(AuxiliaryApplication),
        ( exists( X4 in Type(UserApplication),
                  isCreatedBy_C_C(X3,X4) );
          exists( X4 IN Type(AuxiliaryApplication),
                  isCreatedBy_C_C(X3,X4) ) ),
forall( X5 IN Type(UserApplication),
          exists( X6 IN Interpret(QueryInterpreter),
                  asks_C_M(X5,X6) ) ),
forall( X7 IN Type(UserApplication),
          exists( X8 IN Request(Repository),
                  asks_C_M(X7,X8) ) ),
forall( X9 IN Type(UserApplication),
        exists( X10 IN Type(QueryResult),
                exists( X11 IN Creation(OutputViewer),
                        createsWith_C_C_C(X9,X10,X11) ) ) ).
```

Step 3

In the third and final transformation step we replace the concept and port names by, respectively, the virtual classifications and filters to which they were mapped by the architectural instantiation.

For example, `X1 IN Event(InputWindow)` is replaced by `X1 IN methodFilter(userInput)` because the architectural instantiation maps the **Input Window** concept to the virtual classification `userInput` and the port Event of that concept to a `methodFilter`.

Furthermore, as explained in Subsection 5.3.6, the predicate `filteredIsClassifiedAs(C,F,X)` can be used to generate an artifact `X` that belongs to a virtual classification `C` and satisfies the filter `F`. Therefore, we further transform `X1 IN methodFilter(userInput)` to `filteredIsClassifiedAs(userInput,methodFilter,X1)`.

Applying this transformation process to the entire previous expression in Prolog pseudo-code yields the following Prolog-expression:

```
exists( filteredIsClassifiedAs(userInput,methodFilter,X1),
        ( exists( filteredIsClassifiedAs(userApplication,methodFilter,X2),
                  mentions_M_M(X1,X2) );
          exists( filteredIsClassifiedAs(auxiliaryApplication,methodFilter,X2),
                  mentions_M_M(X1,X2) ) ) ),
forall( filteredIsClassifiedAs(auxiliaryApplication,baseClassFilter,X3),
        ( exists( filteredIsClassifiedAs(userApplication,baseClassFilter,X4),
                  isCreatedBy_C_C(X3,X4) );
          exists( filteredIsClassifiedAs(auxiliaryApplication,baseClassFilter,X4),
                  isCreatedBy_C_C(X3,X4) ) ) ),
forall( filteredIsClassifiedAs(userApplication, baseClassFilter, X5),
        exists( filteredIsClassifiedAs(queryInterpreter, methodFilter, X6),
                asks_C_M(X5,X6) ) ),
forall( filteredIsClassifiedAs(userApplication, baseClassFilter, X7),
        exists( filteredIsClassifiedAs(repository, methodFilter, X8),
                asks_C_M(X7,X8) ) ),
forall( filteredIsClassifiedAs(userApplication,baseClassFilter,X9),
        exists( filteredIsClassifiedAs(result,baseClassFilter,X10),
                exists( filteredIsClassifiedAs(resultViewer,baseClassFilter,X11),
                        createsWith_C_C_C(X9,X10,X11) ) ) ).
```

This concludes the transformation process. Interpreting the obtained Prolog expression above will return either true or false and indicates whether or not the implementation conforms to the 'user interaction' architectural view.

Multiple architectural views

We have illustrated the algorithm only for one single architectural view and not for a complete conceptual architecture consisting of multiple architectural views. The algorithm can be easily generalized to work on a conceptual architecture, by using an expression which is the conjunction of the constructed expressions for each of the architectural views belonging to that conceptual architecture.

This concludes our informal definition of the conformance checking algorithm. In the next subsection, we sketch our Prolog implementation of the conformance checking algorithm.

6.3.2 Implementation

Checking conformance is achieved by means of a predicate `architecturalConformance` which takes the name of an architectural view as input and checks whether the implementation artifacts in the repository conform to the constraints imposed by that architectural view. This is done by checking conformance to all relations in that architectural view.

```
architecturalConformance(ArchView) :-
    forall( relation(ArchView, Relation),
            relationConformance(ArchView, Relation) ).
```

The auxiliary predicate `relationConformance` checks whether the implementation artifacts in the code repository conform to the architectural relation `Relation` in some architectural view `ArchView` and is defined as follows:

```
[01] relationConformance(ArchView, Relation) :-
       % Get the VirtualDependency associated with this Relation
[02]   relationMapping(ArchView, Relation, VirtualDependency),
[03]   findall(    [ Number, Quantifier, Filter, ID ],
                   ( % Find Role belonging to Relation in ArchView.
[04]                 role(ArchView, Relation, Role),
                     % Get the Number associated with this Role.
[05]                 roleMapping(ArchView, Relation, Role, Number),
                     % Find a Port linked with this Role. (multiple links possible)
[06]                 link(ArchView, Concept, Port, Relation, Role),
                     % Get Quantifier associated with this link.
[07]                 linkMapping(ArchView, Concept, Port, Relation, Role, Quantifier),
                     % Get Filter associated with this Port.
[08]                 portMapping(ArchView, Concept, Port, Filter),
                     % Get ID of virtual classification associated with this Concept.
[09]                 conceptMapping(ArchView, Concept, ID)
                     ),
[10]               RoleInfo),
       % Regroup the returned list of quadruples [ Number, Quantifier, Filter, ID ]
       % by number
[11]   regroup(RoleInfo, Regrouped),
       % Apply the predicate VirtualDependency to the Regrouped List.
[12]   virtualApply(VirtualDependency, Regrouped).
```

Let us summarize the implementation of this predicate, without going into all implementation details. On line 2, the predicate looks up the virtual dependency, associated with the `Relation`. Eventually, on line 12, this virtual dependency is checked with the correct arguments. These arguments correspond to the elements in the virtual classifications of the concepts to which the `Relation` is linked. But only those elements that are filtered by the concepts' ports should be considered. To know how exactly the virtual dependency should be applied to the elements of these filtered classifications, the quantifiers associated with the `Relation`'s links need to be known. Therefore, on lines 3 to 10, we accumulate all this information, as well as the argument number, for each of the roles of the relation. (Lines 4 and 5 select the argument `Number`, lines 6 and 7 get the `Quantifier`, line 8 retrieves the port `Filter`, and line 9 finds the `ID` of the virtual classification.)

Now we have the necessary information for applying the virtual dependency. For every argument of the virtual dependency, we know its `Number` as well as the corresponding `Quantifier`, `Filter` and virtual classification `ID`. In line 11, we regroup this information so that it is ordered by increasing role number (`Number`). More precisely, we compile a list of triples of the form `[Quantifier, Filter, ID]`: one for each of the arguments of the virtual dependency predicate. (In fact, there can be more than one triple per argument, as there may be multiple ports associated with the same role. For now, however, we assume that there is only one. We explain later what to do in the case that there are more.) Note that we also drop the role number, as it can be derived from the position in the regrouped list. On line 12, this `Regrouped` list, together with the virtual dependency predicate itself, is then passed to an auxiliary predicate `virtualApply` which actually applies the virtual dependency as explained below.

`virtualApply(VirtualDependency,Regrouped)` applies a predicate `VirtualDependency` to a list `Regrouped`, which is a list of triples of the form `[Quantifier, Filter, ID]`. The idea is that the `Filter` and `ID` of each such triple specify the possible set of values for the corresponding argument of the `VirtualDependencyName` predicate as follows: it is the set obtained by applying the `Filter` on the result of computing the virtual classification with identifier `ID`. The quantifiers are needed to know how the predicate should be applied to these elements. For example,

```
virtualApply(asks_C_M, [ [forall, baseClassFilter, userApplication],
                         [exists, methodFilter, queryInterpreter] ] )
```

boils down to evaluating the following piece of PROLOG code:

```
forall( filteredIsClassifiedAs(userApplication, baseClassFilter, X1),
        exists( filteredIsClassifiedAs(queryInterpreter, methodFilter, X2),
                asks_C_M(X1, X2) ) ).
```

However, there is still a small problem regarding the accumulated role information (`Number`, `Quantifier`, `Filter` and `ID`). As noted above, there may be more than one result for the same role number, which is the case when a role is linked to more than one port. In such a case, as explained in Subsections 5.2 and 5.4.5, we want to take the disjunction over all possible ports linked to that role.

Therefore, the `Regrouped` list produced in line 11 should not be a simple list of triples of the form `[Quantifier, Filter, ID]`, but a nested list of lists of such triples: the first sublist represents all possible triples `[Quantifier, Filter, ID]` corresponding to the first role, the second sublist represents all triples corresponding to the second role, and so on. The `virtualApply` predicate should work with such lists. So, in the above example of the `asks_C_M` predicate, the input should actually be:

```
virtualApply(asks_C_M, [ [ [forall, baseClassFilter, userApplication] ],
                         [ [exists, methodFilter, queryInterpreter] ]
                        ] )
```

A more interesting example (also see 5.4.5) where there are multiple roles associated with the same port is:

```
virtualApply(isCreatedBy_C_C, [ [ [ forall, baseClassFilter, auxiliaryApplication ] ],
                                [ [ exists, baseClassFilter, userApplication ],
                                  [ exists, baseClassFilter, auxiliaryApplication ] ]
                                ])
```

which boils down to evaluating the following piece of PROLOG code:

```
forall( filteredIsClassifiedAs(auxiliaryApplication, baseClassFilter, X1),
        ( exists( filteredIsClassifiedAs(userApplication, baseClassFilter, X2),
                  isCreatedBy_C_C(X1, X2) );
          exists( filteredIsClassifiedAs(auxiliaryApplication, baseClassFilter, X2),
                  isCreatedBy_C_C(X1, X2) )
        ) ).
```

This concludes the sketch of our Prolog implementation of the conformance checking algorithm. In the next subsection, we mention some caching techniques that were used to optimize the efficiency of the algorithm.

6.3.3 Some optimizations

To improve the time-efficiency of the conformance checking algorithm, we incorporate caching techniques in several places. We will not go into the technicalities of how this is implemented.

A first place where caching is used is in the representation of virtual classifications. A virtual classification is computed only the first time, at which point all of its values are cached. The next time it is needed, the values are simply retrieved from the cache. Only when the implementation or the architectural abstraction changes, it is necessary to recompute the virtual classification.

A second place where caching is useful is in the computation of indirect and complex implementation relationships. Whereas some primitive implementation relationships can be retrieved directly from the implementation repository, some frequently occurring higher-level relationships need to be recomputed every time they are needed. A typical example is the transitive closure of the `inheritance` relationship between classes (i.e., checking whether some class belongs to the inheritance `hierarchy` of another one). Because this derived relation (i.e., `hierarchy`) is needed so often, we compute and cache all such relationships beforehand, so that they can be simply retrieved later when they are needed. The same is done for other frequently-needed relationships such as `understands`.

Other places where caching can be used and other techniques to optimize the time-efficiency of the algorithm (as well as other optimizations) are discussed in Section 8.2.

6.4 Extending the architectural formalism

The architectural formalism that was proposed in Chapter 5 is still very primitive. Consider for example the proposed ADL: it only contains a notion of concepts and relations with ports and roles that are connected by links. It does not (yet) allow us to express sub-architectures, nor does it provide support for architectural styles and patterns. We consider these extensions as important future work. In this section we take a closer look at these and other extensions and explain where and how the architectural formalism should be updated to accommodate them.

6.4.1 Refined notation

The proposed architecture language is very flexible and expressive. One could even argue that, in some sense, it may be even too expressive. For example, consider some architectural view that is described in our general ADL. Such an architectural view, especially in its graphical form, is very important because it serves as a communication element between the members of a software project. Its main purpose is precisely to provide a simple mental picture that allows software engineers to quickly grasp the global structure of a software system. Therefore, just by looking at the picture one should obtain some kind of intuition about the software structure.

Unfortunately, with the current generality of the architecture language, it is not always possible to 'understand' a system from the diagram only, without having to resort to the other levels of the architecture language, i.e., the architectural instantiation and the architectural abstraction. For example, suppose we have an architectural concept which is mapped to a virtual classification consisting of classes. What does this mean? Does each class in the classification implement that concept or do the classes taken collectively implement the concept? (In fact, both answers are possible. For example, in the 'user interaction' architectural view, the **User Application** concept is mapped to a set of classes that, each on their own, represent a different kind of user application. The **Query Result** concept in the same architectural view, however, is mapped to a set of classes that together represent the result of a query. One main class uses the state design pattern to distribute its work over several auxiliary classes.) The current ADL does not allow us to answer important questions such as these; the declarations in the AML need to be considered as well.

Therefore, it is important future work to refine the ADL (and its corresponding graphical notation) to make this intuition more clear. Depending on the kind of semantics intended, a different notation could be used. For example, we could have a special denotation for architectural concepts that represent some generic concept that is mapped to a set of classes, each of which implements a specific variant of that concept. Architectural concepts that are mapped to a set of classes that collectively represent the concept could be depicted by another notation. Using such a more refined notation, an architectural view would give a better insight in the intended semantics, without forcing us to take the architectural mapping into account.

In addition, a more refined ADL has some other advantages as well. First of all, the different notations may have different constraints associated with them. For example, suppose again that one has an architectural concept which is mapped to a set of classes that each implement the concept. This concept will typically need a 'Type' port to be associated with it, to retrieve the different classes that represent the concept. Such a port may have little or no use for other kinds of concepts. Secondly, the different notations also impose constraints on the allowed architectural mappings. For example, a certain kind of concepts will typically mapped to the implementation according to a certain mapping scheme (there may be multiple alternative mapping schemes) whereas another kind of concepts may have other default mapping schemes associated with it. Also, if we have a concept mapped to a set of classes that each implement the concept, we know not only that the concept should probably have a 'Type' port, but we also know that this port should be mapped to a class filter.

To accommodate all this, the following extensions to the architecture language are needed:

- Extend the ADL with specialized notations, allowing different kinds of architectural concepts, relations, ports, roles, etc.

- Extend the ADL so that it supports the declaration and enforcement of architectural constraints on these new kinds of architectural entities.

- Extend the (architectural layer of the) DFW with an extra layer describing the typical kinds of architectural mappings for each of the new kinds of architectural entities.

- Extend the AML to allow the declaration and enforcement of constraints on how architectural entities are mapped to the implementation. Positive constraints may indicate which mapping scheme to choose. Negative constraints may exclude certain mapping schemes.

6.4.2 Architectural styles

Our current architectural language provides no support for *architectural styles*. Adding this feature essentially requires the same extensions to the architectural language as those discussed in the previous subsection. To adequately support architectural styles, we need customized notations, specific constraints for each of those notations, default mapping schemes for the different kinds of architectural entities in a certain style, and constraints on the instantiation of certain entities with certain architectural mappings.

In Subsection 2.1.2, we mentioned the 'pipe and filter' and 'pipeline' styles as particular examples of architectural styles. A style such as a 'pipeline' may require customized notations to represent 'pipes' and 'filters', and may declare some constraints on how the different kinds of architectural entities can (and cannot) be interconnected in an architectural view that complies to this style. (Examples of constraints in the 'pipeline' architectural style are: 'filter' concepts can only be connected through 'pipe' relations; pipes connect exactly two ports; there can be no 'dangling' pipes; ports of filters can connect to no more than one pipe; etc.) Furthermore, default mappings could be associated with 'pipes' and 'filters', representing the typical ways how they could be implemented in the base language. Again, these default mappings should be put in an extra architectural layer of the DFW which groups all mappings that are specific to a certain architectural style. Constraints can be used to declare which kinds of style-specific architectural entities can and cannot be instantiated with which mappings.

As a concrete example of a structural constraint that is imposed by a certain architectural style, we repeat an experiment from an earlier paper.[4] More precisely, we define a parameterized predicate that describes the general structure of a 'pipeline' architecture, which is composed out of filter concepts and pipe relations. It implements a complex architectural pattern that uses LMP to declare the architectural configuration of all involved architectural relations and concepts.

```
pipeFilterPattern(ArchView, [Filter1, Filter2 | OtherFilters], [Pipe1 | OtherPipes]) :-
    link(ArchView, Filter1, out, Pipe1, source),
    link(ArchView, Filter2, in, Pipe1, target),
    % recursive call:
    pipeFilterPattern(ArchView, [Filter2 | OtherFilters], OtherPipes).

% end of recursion:
pipeFilterPattern(ArchView, [LastFilter], []).
```

The first argument defines the scope (i.e. the current architectural view) in which we are working. When supplied with a list of filters (second argument) and a list of pipes (third argument), the predicate checks whether the pipes connect the filters, in respective order. In other words, the first filter should be linked to the second filter by the first pipe, the second filter to the third filter by the second pipe, and so on.

[4] At the time of writing this dissertation, the experiment has not yet been tried out in the current version of our architectural formalism. It was tested, however, in a slightly older version [52] and is presented here using our current notations.

6.4.3 Architectural correspondence

We agree with Kruchten [38] that the various architectural views on a software implementation need not be fully independent. Elements of one view may be connected to elements in other views, following certain rules. Just like we codified the mapping between the implementation and the different architectural views in a logic meta language, we could also codify the *correspondence* between (elements of) the various architectural views. To this extent, the ADL should be extended to allow the declaration of such correspondences and the conformance checking algorithm should be extended to verify these correspondences as well.

Because we use a full-fledged LMP language, we have no doubt that it is possible to define (and verify) these correspondences in that language. However, some research should be put into what kinds of *architectural correspondences* are possible and useful, as well as how they should be modeled and represented in our conformance checking formalism, and how they could be verified. Below, we give an indication of some possible kinds of architectural correspondences.

Correspondences among concepts relate concepts in different architectural views. Two kinds of such correspondences may be distinguished: those that can be represented as architectural relations among concepts, and those that cannot. The first kind of correspondences allow to reason only about the artifacts that belong to the virtual classifications to which the concepts are mapped. The second kind of correspondences require extra information about the concepts such as their ports or the name of the virtual classifications with which they are instantiated. Examples of the first kind are:

- Requiring that two concepts have the same extension, i.e., the virtual classifications to which they are mapped compute the same set of artifacts.
- Requiring that two concepts have a disjoint extension.
- Requiring that two concepts have an overlapping extension.
- Declaring that a concept is a strengthening or weakening of another one (in the sense that their classifications are in a subset or superset relationship).
- Expressing other architectural relations between concepts in different architectural views. For example, we could assert that there should be a *Creates* relationship between the **User Application** concept from the 'user interaction' view and the **Query** concept from the 'application architecture' view.

Examples of correspondences between architectural concepts that cannot be expressed by ordinary architectural relations are:

- Requiring that two concepts have the same intention, i.e., they are mapped to exactly the same virtual classification. (Of course, this implies that their extension will also be the same.) An example of this is the correspondence between the **Repository** concept in the 'user interaction' view and the **Knowledge Base** concept in the 'rule-based interpreter' view, or the **Query Interpreter** concept in the 'user interaction' view and the **Rule Interpreter** concept in the 'rule-based interpreter' view.
- Requiring that two concepts have the same set of ports; or a disjoint set of ports; or an overlapping set of ports; or checking for a subset relationship between the set of ports.

Correspondences among relations declare some relationship between architectural relations in different architectural views. For example, we could declare that two architectural relations are mapped to the same virtual dependency.

Other correspondences may include correspondences among ports, among roles, or even correspondences among architectural entities of different kinds such as concepts and relations. (We might even want to express correspondences among correspondences.)

The correspondences among concepts that can be expressed by means of architectural relations can easily be incorporated in our formalism. For the other kinds of architectural correspondences, we need to investigate in detail how they can best be modeled and verified in our formalism.

6.4.4 Architectural deviations

Although changes to the implementation that do not conform to the architecture should be avoided, sometimes this is not possible (due to technical problems, time pressure, etc.). In such a situation these *deviations* from the architecture should be annotated explicitly and should be taken into account during the conformance checking process. For example, when checking conformance to a certain architectural relation, the relation should be verified for all relevant artifacts, except for the deviations.

6.4.5 Sub-architectures

As will be illustrated in Chapter 7, our architecture language supports the definition of composite virtual classifications and composite virtual dependencies that are defined in terms of more primitive virtual classifications and virtual dependencies. However, the architecture language does not yet support the definition of *composite architectural concepts* or *composite architectural relations* that are defined in terms of sub-architectures consisting of more primitive architectural concepts and relations and their interconnections. As this was one of our expressiveness requirements (see 3.2.2), in this subsection we explain in short how the architecture language and conformance checking algorithm could be extended to deal with such composite entities.

Composite concepts

As in Subsection 6.4.2, we illustrate this extension of the architectural language by repeating an experiment from an earlier paper.[5] More precisely, we will describe the computational process of the **Rule Interpreter** concept in the 'rule-based interpreter' view in terms of a sub-architecture (instead of directly in terms of a virtual classification).

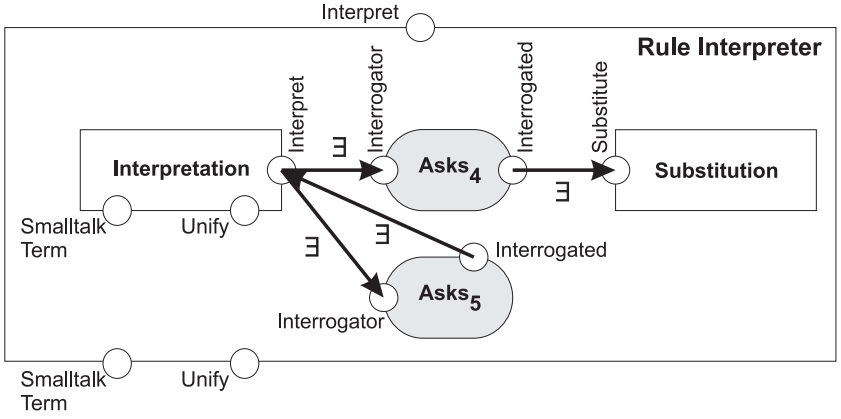

Figure 6.4: The 'Rule Interpreter' sub-architecture.

We base our description of this sub-architecture on the insight that interpreting a rule actually proceeds in two steps: an *interpretation* phase where all terms and clauses occurring in the rule are interpreted recursively, and a *substitution* phase where bindings found during unification are

[5] At the time of writing the dissertation, this extension has not yet been included in the current version of our architecture language and conformance checking algorithm. It was implemented, however, in a slightly older version. The experiment presented here is a retake of an experiment in that older version [52] and was adapted to our current notations.

substituted in the term that is currently being interpreted. We define this sub-architecture in terms of two architectural concepts **Interpretation** and **Substitution**, connected by means of *Asks* relations, as illustrated in Figure 6.4. The $Asks_4$ relation represents the fact that substitutions are performed during the interpretation phase, and the $Asks_5$ relation represents the recursive nature of the interpretation phase.

The sub-architecture of Figure 6.4 is described by the facts listed in Table 6.1. There is one fact for each concept, relation, port, role and link. The only difference with the description of an architectural view, is that instead of a `view` fact, we need to declare a `subArchitecture` fact. Whereas the `view` fact defines the name of an architectural view and the conceptual architecture it belongs to, the `subArchitecture` fact declares the name of a sub-architecture and the name and view of the concept it is defining.

```
subArchitecture(soulRuleBasedSystem, ruleInterpreter, ruleInterprSubArch).
concept(ruleInterprSubArch, interpretation).
concept(ruleInterprSubArch, substitution).
relation(ruleInterprSubArch, asks4).
relation(ruleInterprSubArch, asks5).
port(ruleInterprSubArch, interpretation, interpret).
port(ruleInterprSubArch, interpretation, smalltalkTerm).
port(ruleInterprSubArch, interpretation, unify).
port(ruleInterprSubArch, substitution, substitute).
role(ruleInterprSubArch, asks4, interrogator).
role(ruleInterprSubArch, asks4, interrogated).
role(ruleInterprSubArch, asks5, interrogator).
role(ruleInterprSubArch, asks5, interrogated).
link(ruleInterprSubArch, interpretation, interpret, asks4, interrogator).
link(ruleInterprSubArch, substitution, substitute, asks4, interrogated).
link(ruleInterprSubArch, interpretation, interpret, asks5, interrogator).
link(ruleInterprSubArch, interpretation, interpret, asks5, interrogated).
```

Table 6.1: Architectural description of the 'Rule Interpreter' sub-architecture.

Because the **Rule Interpreter** concept is now described by the sub-architecture shown in Figure 6.4 and Table 6.1 (instead of by some virtual classification), we do not need to declare a concept mapping for that concept anymore. Instead, the meaning of the **Rule Interpreter** concept will be derived from the architectural mappings associated with each of the entities in its sub-architecture. These architectural mappings are shown in Table 6.2.

For example, the **Interpretation** concept is mapped to the virtual classification 'queryInterpreter' in terms of which the **Rule Interpreter** concept was originally defined. The **Substitution** concept is mapped to a virtual classification 'substitution' containing all methods that belong to a method protocol named 'substitution'. It also contains all classes that implement one of these methods. This virtual classification is defined by the rules below:

```
classifiedAs(method('substitution'), Method) :-
    classifiedAs(class('soul'), Class),   % restrict scope to 'SOUL' classes
    protocolName(Protocol, 'substitution'),
    methodInProtocol(Class, Protocol, Method).

classifiedAs(class('substitution'), Class) :-
    findClassesFromMethods(Class, 'substitution').
```

Just like the meaning of the composite **Rule Interpreter** concept will be derived from the architectural instantiation of its sub-architecture, the meaning of the external ports of the **Rule Interpreter** concept, will be defined in terms of the meanings of the internal ports (i.e., ports belonging to concepts within the sub-architecture). We do not need to declare a port mapping for

```
conceptMapping(ruleInterprSubArch, interpretation, queryInterpreter).
conceptMapping(ruleInterprSubArch, substitution, substitution).
portMapping(ruleInterprSubArch, interpretation, interpret, methodFilter).
portMapping(ruleInterprSubArch, interpretation, unify, methodFilter).
portMapping(ruleInterprSubArch, interpretation, smalltalkTerm, smalltalktermClassFilter).
portMapping(ruleInterprSubArch, substitution, substitute, methodFilter).
relationMapping(ruleInterprSubArch, asks4, asks_M_M).
relationMapping(ruleInterprSubArch, asks5, asks_M_M).
roleMapping(ruleInterprSubArch, asks4, interrogator, 1).
roleMapping(ruleInterprSubArch, asks4, interrogated, 2).
roleMapping(ruleInterprSubArch, asks5, interrogator, 1).
roleMapping(ruleInterprSubArch, asks5, interrogated, 2).
linkMapping(ruleInterprSubArch, interpretation, interpret, asks4, interrogator, exists).
linkMapping(ruleInterprSubArch, substitution,  substitute, asks4, interrogated, exists).
linkMapping(ruleInterprSubArch, interpretation, interpret, asks5, interrogator, exists).
linkMapping(ruleInterprSubArch, interpretation, interpret, asks5, interrogated, exists).
```

Table 6.2: Architectural instantiation for the 'Rule Interpreter' sub-architecture.

the external ports of **Rule Interpreter** anymore. Instead, each external port should be bound to one (or more) internal port(s). Figure 6.5 and Table 6.3 show these *bindings* for the **Rule Interpreter** concept. (Note that we left out the architectural relations and links in the figure, to avoid cluttering the figure.)

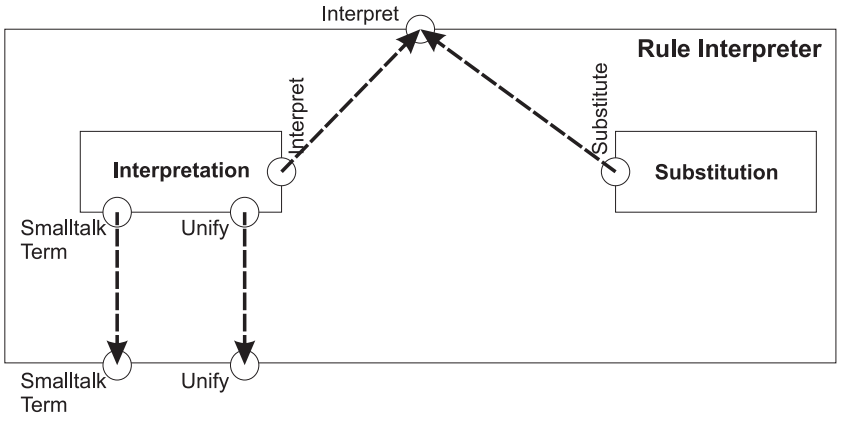

Figure 6.5: Bindings for the 'Rule Interpreter' sub-architecture.

```
portBinding(soulRuleBasedSystem, ruleInterpreter, interpret,
            ruleInterprSubArch, interpretation, interpret).
portBinding(soulRuleBasedSystem, ruleInterpreter, interpret,
            ruleInterprSubArch, substitution, substitute).
portBinding(soulRuleBasedSystem, ruleInterpreter, unify,
            ruleInterprSubArch, interpretation, unify).
portBinding(soulRuleBasedSystem, ruleInterpreter, smalltalkTerm,
            ruleInterprSubArch, interpretation, smalltalkTerm).
```

Table 6.3: Port bindings for the 'Rule Interpreter' sub-architecture.

This concludes the definition of the composite concept **Rule Interpreter** in terms of a sub-architecture. The next section informally explains how the conformance checking algorithm uses all this information to check conformance for such a composite concept.

Checking conformance to composite concepts

We explain how the original conformance checking algorithm should be updated to work in the current situation; where concepts can be instantiated with sub-architectures that are, in turn, built up from many other concepts and relations.

1. First of all, in addition to checking conformance to every architectural view, the algorithm should check conformance to every sub-architecture as well. Indeed, the architectural relations in sub-architectures represent additional constraints that should be satisfied by the implementation.

2. Secondly, whenever the values on an external port of a composite concept are needed, the port bindings need to be taken into account as follows: the set of all values corresponding to an external port of a composite concept is the union of all values corresponding to each of the internal ports to which this external port is bound. (If the composite concept would be defined in terms of a sub-architecture which in turn contains another composite concept, we simply apply this rule recursively.)

Composite relations

In Subsection 7.3.3, we will show how the *Is Composite* architectural relations are defined directly in terms of a virtual dependency `isComposite_C_C`, which is a conjunction of the virtual dependencies `specializes_C_C` and `containsElementsOfType_C_C`. However, just like we can define composite concepts, it is possible to define composite relations. Therefore, alternatively we could define *Is Composite* as a composite architectural relation in terms of the more primitive relations *Is Kind of* and *Contains*, as illustrated in Figure 6.6.

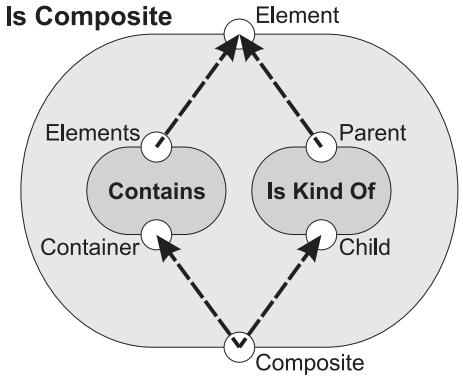

Figure 6.6: A composite architectural relation: 'Is Composite'.

To define such a composite relation, a similar approach as for defining a composite concept should be followed. First of all, we need to describe all entities of the sub-architecture and their architectural mapping. For the composite relation itself, we do not need to define a relation mapping anymore. Also, for its (external) roles, we do not need to define role mappings anymore. We do need to declare *role bindings* to bind every external role of the composite relation to one or more roles of internal relations (i.e., relations belonging to relations within the sub-architecture).

Checking conformance to composite relations

In order to be able to handle composite relations, the conformance checking algorithm should be updated as follows:

1. In addition to checking conformance to all architectural views and all sub-architectures of composite concepts, the algorithm should also check conformance to all sub-architectures of composite relations.

2. Whenever a composite relation is checked with certain values, the role bindings are used to propagate the values to the corresponding roles of internal relations, and these internal relations should be checked instead.

6.5 Summary

In Chapter 5, we introduced and formalized the architecture language and defined a notion of architectural conformance. In this chapter, we explained how to implement this architectural formalism in a LMP language. This was very easy and straightforward, due to the declarative nature and expressive power of LMP, and due to the 'logic flavor' of the architectural formalism.

We sketched the setup of our LMP environment, showed how to represent the architecture language(s) in this environment and presented the implementation of an architectural conformance checking algorithm in the LMP language. We concluded the chapter with a discussion of some extensions to the architectural formalism.

Chapter 7

Case Study

In this chapter, we elaborate on the case study we conducted as a validation of the thesis. We illustrate how we checked conformance of the implementation of SOUL to the different architectural views presented in Chapter 4. This case study proves the validity of our approach and shows that combining logic meta programming with the notions of virtual classifications and virtual dependencies offers a very expressive medium for checking architectural conformance.

7.1 The user interaction architectural view

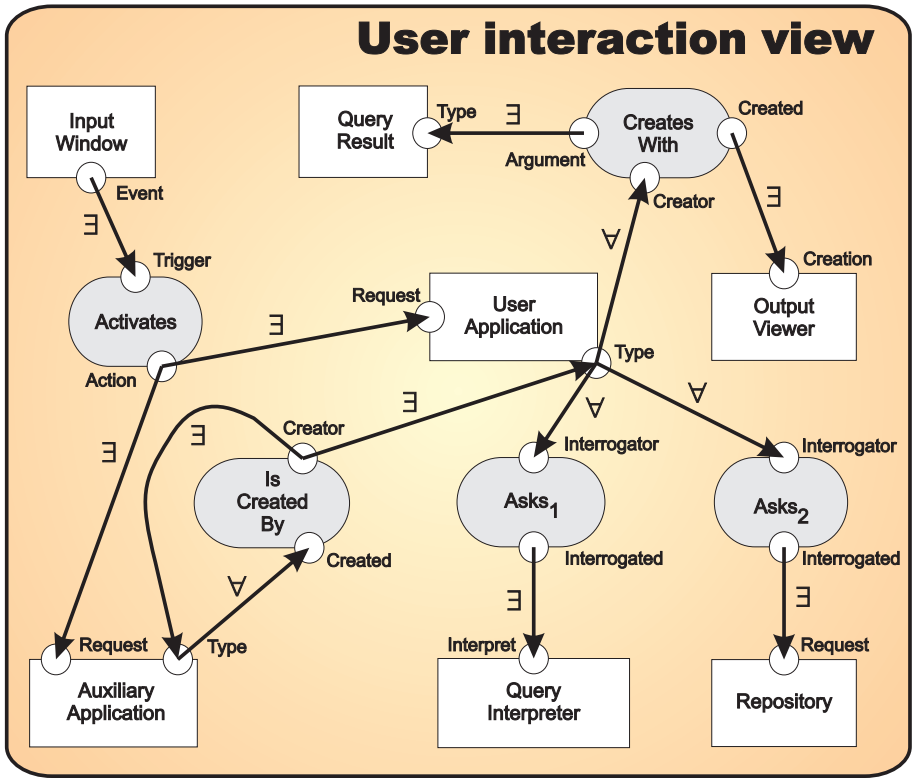

Figure 7.1: The 'user interaction' architectural view with quantifiers.

Now that we have explained the architectural formalism and how it was implemented in a LMP language, we turn our attention to the conducted case study. For each of the architectural

views presented in Chapter 4, we take a closer look at how the architectural mapping was defined, we discuss some of the difficulties encountered, and we illustrate why our approach is indeed an expressive one. In this particular section, we elaborate on our experiences with checking conformance of the Smalltalk implementation of SOUL to the 'user interaction' architectural view of Section 4.2.2. For easy reference, we repeat Figure 5.2 of the 'user interaction' view here (Figure 7.1).

7.1.1 Declaring the user interaction architectural view

As explained in Subsection 6.2.1, we describe architectural views by declaring the entities they contain as a set of Prolog facts. All these facts together describe the 'user interaction' architectural view. Table 7.1 shows some of the facts for the 'user interaction' architectural view depicted in Figure 7.1. Some facts have been left out and are indicated by '...'.

```
view(soul, soulUserInteraction).
...
concept(soulUserInteraction, inputWindow).
concept(soulUserInteraction, userApplication).
concept(soulUserInteraction, queryInterpreter).
...
relation(soulUserInteraction, activates).
relation(soulUserInteraction, asks1).
...
port(soulUserInteraction, inputWindow, event).
port(soulUserInteraction, userApplication, request).
port(soulUserInteraction, userApplication, type).
port(soulUserInteraction, queryInterpreter, interpret).
...
role(soulUserInteraction, activates, trigger).
role(soulUserInteraction, activates, action).
role(soulUserInteraction, asks1, interrogator).
role(soulUserInteraction, asks1, interrogated).
...
link(soulUserInteraction, inputWindow, event, activates, trigger).
link(soulUserInteraction, userApplication, request, activates, action).
link(soulUserInteraction, userApplication, type, asks1, interrogator).
link(soulUserInteraction, queryInterpreter, interpret, asks1, interrogated).
...
```

Table 7.1: Declaring the 'user interaction' architectural view.

7.1.2 Declaring the architectural instantiation

In Subsection 6.2.3, we explained how an architectural instantiation is represented in our LMP language. Tables 7.2 to 7.6 present the complete architectural instantiation for the 'user interaction' architectural view of Figure 7.1.

Table 7.2 lists the facts that define the concept mappings. Note that in many cases the concept name and the name of the associated virtual classification (second and third argument) are the same. This is not a problem, because they are stored separately in the repository.

The port mappings for the concepts in the 'user interaction' view are shown in Table 7.3. The relation mappings are listed in Table 7.4, and Table 7.5 presents the role mappings for the 'user interaction' architectural view. Finally, the link mappings for this architectural view are given in Table 7.6.

```
conceptMapping(soulUserInteraction, inputWindow,          userInput).
conceptMapping(soulUserInteraction, userApplication,      userApplication).
conceptMapping(soulUserInteraction, auxiliaryApplication, auxiliaryApplication).
conceptMapping(soulUserInteraction, outputViewer,         resultViewer).
conceptMapping(soulUserInteraction, queryInterpreter,     queryInterpreter).
conceptMapping(soulUserInteraction, repository,           repository).
conceptMapping(soulUserInteraction, queryResult,          result).
```

Table 7.2: Concept mappings for the 'user interaction' architectural view.

```
portMapping(soulUserInteraction, inputWindow,          event,     methodFilter).
portMapping(soulUserInteraction, outputViewer,         creation,  baseClassFilter).
portMapping(soulUserInteraction, userApplication,      request,   methodFilter).
portMapping(soulUserInteraction, userApplication,      type,      baseClassFilter).
portMapping(soulUserInteraction, repository,           request,   methodFilter).
portMapping(soulUserInteraction, queryInterpreter,     interpret, methodFilter).
portMapping(soulUserInteraction, queryResult,          type,      baseClassFilter).
portMapping(soulUserInteraction, auxiliaryApplication, request,   methodFilter).
portMapping(soulUserInteraction, auxiliaryApplication, type,      baseClassFilter).
```

Table 7.3: Port mappings for the 'user interaction' architectural view.

```
relationMapping(soulUserInteraction, activates,    mentions_M_M).
relationMapping(soulUserInteraction, createsWith,  createsWith_C_C_C).
relationMapping(soulUserInteraction, asks2,        asks_C_M).
relationMapping(soulUserInteraction, isCreatedBy,  isCreatedBy_C_C).
relationMapping(soulUserInteraction, asks1,        asks_C_M).
```

Table 7.4: Relation mappings for the 'user interaction' architectural view.

```
roleMapping(soulUserInteraction, activates,   trigger,      1).
roleMapping(soulUserInteraction, activates,   action,       2).
roleMapping(soulUserInteraction, isCreatedBy, created,      1).
roleMapping(soulUserInteraction, isCreatedBy, creator,      2).
roleMapping(soulUserInteraction, asks1,       interrogator, 1).
roleMapping(soulUserInteraction, asks1,       interrogated, 2).
roleMapping(soulUserInteraction, asks2,       interrogator, 1).
roleMapping(soulUserInteraction, asks2,       interrogated, 2).
roleMapping(soulUserInteraction, createsWith, creator,      1).
roleMapping(soulUserInteraction, createsWith, argument,     2).
roleMapping(soulUserInteraction, createsWith, created,      3).
```

Table 7.5: Role mappings for the 'user interaction' architectural view.

7.1.3 Virtual classifications

Now that we have presented the implementation of the architecture description and architectural instantiation of the 'user interaction' architectural view, we show how the architectural abstraction is represented. We explain this for each of the different kinds of architectural abstractions separately, starting with the virtual classifications.

```
linkMapping(soulUserInteraction,inputWindow,event,activates,trigger,exists).
linkMapping(soulUserInteraction,userApplication,request,activates,action,exists).
linkMapping(soulUserInteraction,auxiliaryApplication,request,activates,action,exists).
linkMapping(soulUserInteraction,auxiliaryApplication,type,isCreatedBy,created,forall).
linkMapping(soulUserInteraction,userApplication,type,isCreatedBy,creator,exists).
linkMapping(soulUserInteraction,auxiliaryApplication,type,isCreatedBy,creator,exists).
linkMapping(soulUserInteraction,userApplication,type,asks1,interrogator,forall).
linkMapping(soulUserInteraction,queryInterpreter,interpret,asks1,interrogated,exists).
linkMapping(soulUserInteraction,userApplication,type,asks2,interrogator,forall).
linkMapping(soulUserInteraction,repository,request,asks2,interrogated,exists).
linkMapping(soulUserInteraction,userApplication,type,createsWith,creator,forall).
linkMapping(soulUserInteraction,queryResult,type,createsWith,argument,exists).
linkMapping(soulUserInteraction,outputViewer,creation,createsWith,created,exists).
```

Table 7.6: Link mappings for the 'user interaction' architectural view.

As a first and simple example of a virtual classification, we define the 'userApplication' virtual classification which is used to model the architectural concept **User Application** in the SOUL system. This virtual classification is fairly easy to define thanks to the coding conventions adopted by the developers of the SOUL system. In particular, all user applications (as well as some auxiliary applications) were implemented as Smalltalk classes that are stored together in the same class category 'SOULUIApplications'. Furthermore, the *naming convention* was adopted to end the name of every class representing a user application with 'App'. Also, as for all other classes in the SOUL implementation, these classes start with the string 'SOUL'. All this is codified by the Prolog predicate below. The predicate head should be read as: "the virtual classification `userApplication` contains those elements `Class` of type `class`, that satisfy ...".

```
classifiedAs(class('userApplication'), Class) :-
    categoryName(Category, 'SOULUIApplications'),
    classInCategory(Category, Class),
    className(Class, ClassName),
    patternMatch(ClassName, and(prefix('SOUL'), postfix('App'))).
```

Such an intentional and declarative definition has many advantages over a more extensional (i.e., enumerating) one (also see Subsection 2.3.4). First of all, it contains more knowledge, because it makes some of the developer's assumptions explicit. Secondly, it is more compact than an exhaustive enumeration of all elements. And finally, it is more reusable and robust towards evolution than an extensional definition. For example, if the developer would want to add new user applications while respecting the same coding conventions, after this modification, the intentionally defined virtual classification would still correctly define its elements.

In addition to artifacts of type class, the virtual classification 'userApplication' also contains artifacts of type method, representing the requests that can be handled by user applications. In other words, this classification is an example of a *heterogeneous* virtual classification containing a mixture of both Smalltalk classes and methods. The methods are defined straightforwardly in terms of the already declared class elements.

```
% 'userApplication' methods are methods that belong to 'userApplication' classes
classifiedAs(method('userApplication'), Method) :-
    findMethodsFromClasses(Method, 'userApplication').
```

A second example of a heterogeneous virtual classification is the 'queryInterpreter' virtual classification which groups all classes and methods that have something to do with the interpretation of queries in the SOUL system. Again, we are fortunate, because the SOUL developers adopted a *coding convention* to store all query-interpretation methods in the same Smalltalk method protocol 'interpretation', 'interpreting' or 'unification'.

```
classifiedAs(method('queryInterpreter'), Method) :-
    classifiedAs(class('soul'), Class),        % restrict the scope to  'SOUL'  classes
    interpretingProtocolName(ProtocolName),    % select  interpretation method protocol
    protocolName(Protocol, ProtocolName),
    methodInProtocol(Class, Protocol, Method). % extract the methods from this protocol
% Auxiliary predicate:
interpretingProtocolName('interpretation').
interpretingProtocolName('interpreting').
interpretingProtocolName('unification').
```

This classification defines a real *cross-cut* of the Smalltalk code of the SOUL system, as the interpretation methods are distributed over many different classes in the SOUL class hierarchy. (Almost every class representing a node in the SOUL abstract grammar implements one or more of these methods.)

The classes that belong to the 'queryInterpreter' virtual classification can be computed straightforwardly from the already declared method elements. This is the opposite situation of the previous example, where the classified methods were defined in terms of the already declared class elements.

```
% 'queryInterpreter' classes are all classes that implement a 'queryInterpreter' method.
classifiedAs(class('queryInterpreter'), Class) :-
    findClassesFromMethods(Class, 'queryInterpreter').
```

Thanks to our use of a LMP language and the predefined logic predicates provided by the DFW, both virtual classifications above are codified in a *concise, intuitive and readable* way. As a further example of the *expressive power* of using a LMP language, consider the virtual classification 'auxiliaryApplication' for the **Auxiliary Application** concept, in which we make explicit use of logic *negation*. The Smalltalk category 'SOULUIApplications' groups all classes representing user applications, auxiliary applications and output viewers. Hence, auxiliary applications correspond to those classes that belong to this category and that are *not* a 'userApplication' class, *nor* a 'resultViewer' class[1].

```
classifiedAs(class('auxiliaryApplication'), Class) :-
    categoryName(Category, 'SOULUIApplications'),
    classInCategory(Category, Class),
    not classifiedAs(class('userApplication'),Class),
    not classifiedAs(class('resultViewer'),Class).
```

Similar to the 'userApplication' classification, the virtual classification for auxiliary applications also contains methods, and these methods can be derived straightforwardly from the already declared class elements.

```
classifiedAs(method('auxiliaryApplication'), Method) :-
    findMethodsFromClasses(Method, 'auxiliaryApplication').
```

In addition to illustrating the use of *negation* in virtual classifications, the previous example also shows how virtual classifications can be *defined in terms of other virtual classifications*: 'auxiliaryApplication' classes were defined in terms of the classes classified as 'userApplication' or 'resultViewer'.

The 'resultViewer' classification corresponds to the **Output Viewer** architectural concept and is defined as follows:

```
classifiedAs(class('resultViewer'), Class) :-
    hierarchy(class('SOULFindResultPresenter',_), Class);
    hierarchy(class('SOULResultPresenterInspector',_), Class).
```

[1] 'resultViewer' is the name of the virtual classification associated with the **Output Viewer** concept.

In other words, the 'resultViewer' classes are the SOUL classes `SOULFindResultPresenter` and `SOULResultPresenterInspector` and their subclasses. For now, we simply *enumerate* these possibilities.

The architectural concept **Query Result** is mapped to a virtual classification 'result' which groups all classes that implement the results of queries in the SOUL system. This is essentially the `SOULResultPresenter` class. However, this class delegates part of its data and behavior to some auxiliary classes such as `SOULResultState` and its subclasses. Fortunately, in the SOUL implementation, the convention was adopted to prefix the name of all these classes with the string '`SOULResult`'. To reduce the scope, we additionally require that these classes belong to the `SOULObject` class hierarchy.

```
classifiedAs(class('result'), Class) :-
    hierarchy(class('SOULObject',_), Class),
    className(Class, ClassName),
    patternMatch(ClassName, prefix('SOULResult')).
```

The **Input Window** concept is defined in terms of a virtual classification 'userInput' which groups all methods representing events that can be triggered by users, i.e., all methods for handling window buttons, fields or menus. Again, this can be defined straightforwardly based on the following coding conventions: methods associated with window buttons and fields are typically stored in a method protocol 'interface specs', and methods associated with window menus in a protocol named 'resources'.

```
classifiedAs(method('userInput'), Method) :-
    userInputProtocol(ProtocolName),
    methodInProtocol(_, methodProtocol(ProtocolName, _), Method).
userInputProtocol('interface specs'). % methods associated with window buttons and fields
userInputProtocol('resources').       % methods associated with window menus
```

Finally, we have the **Repository** concept which is mapped to the classification 'repository':

```
% 'repository' classes are subclasses of the class SOULAbstractRepository.
classifiedAs(class('repository'), Class) :-
    hierarchy(class('SOULAbstractRepository', _), Class).

% 'repository' meta classes are the meta classes of 'repository' classes.
classifiedAs(metaclass('repository'), Meta) :-
    findMetaClassesFromClasses(Meta, 'repository').

% 'repository' methods belong to 'repository' classes or metaclasses,
classifiedAs(method('repository'), Method) :-
    findMethodsFromClasses(Method, 'repository').
% OR are Factory Methods that create 'repository' classes
classifiedAs(method('repository'), Method) :-
    findMethodsFromClasses(Method, 'soul'), % restrict scope to SOUL classes
    classifiedAs(class('repository'), Class),
    factoryMethod(Method, Class).
```

In the SOUL implementation, repositories are represented by the class `SOULAbstractRepository` or one of its subclasses. The virtual classification for **Repository** contains these classes, as well as all their meta classes and all their methods. Furthermore, in addition to these methods, we also include all methods in the SOUL implementation that are *factory methods* for repository classes. Indeed, the SOUL implementation uses the Factory Method design pattern where new instances of repository classes are always created through the appropriate factory methods (which are all defined on a factory class `SOULFactory`). As these factory methods represent the interface to create new repositories, we include them in the 'repository' classification (even though they are not implemented by a repository class).

The above examples illustrate, amongst others, the ability to define heterogeneous virtual classifications; virtual classification based on naming conventions, coding conventions and design patterns; the use of semantic inferencing; the ability to define virtual classifications in terms of other virtual classifications and virtual dependencies; the use of logic negation; and so on. Furthermore, most rules defining virtual classifications are surprisingly concise, intuitive and readable. In the next sections, more examples will follow.

7.1.4 Port filters

The architectural layer of the DFW provides a whole range of predefined port filters (see 6.2.4), such as method filters and class filters. These predefined filters are sufficient to define most port mappings.

Ports representing actions (an event that is sent, a request that can be handled, the interpretation of a query, ...) are typically defined in terms of a method filter. For example, the **Query Interpreter** concept in the 'user interaction' view has an Interpret port which selects all methods that take part in the interpretation process. So the filter associated with this port is a method filter which selects only the methods from the classification associated with **Query Interpreter**.

Ports representing the type or kind of things (a type of user or auxiliary application, a kind of output viewer that can be created, a kind of query result, ...) are typically defined in terms of (base) class filters.

Table 7.3 lists all port mappings for the 'user interaction' view on the SOUL implementation.

7.1.5 Virtual dependencies

Some of the virtual dependencies required by the 'user interaction' architectural view, were already explained in Subsection 6.2.4. The `mentions_M_M` predicate, which implements the *Activates* architectural relation, was defined in terms of simple *string pattern matching*. The `asks_C_M` predicate, which implements the $Asks_1$ and $Asks_2$ relations, uses a mixture of *pattern matching* and *parse-tree traversing*.

The *Creates With* architectural relation has three roles and is defined in terms of a ternary predicate `createsWith_C_C_C` which checks whether some `SourceClass` creates a `ViewClass` to view an instance of some class `Type`. The predicate distinguishes two cases, codifying the different ways in which this creation behavior can be achieved. The first case verifies whether `SourceClass` contains a method which can create an instance of the `ViewClass` (by sending some instance-creation message directly to this class), and whether this creation method takes an argument of class `Type`. As an example, consider the definition of the following Smalltalk method:

```
showResult:  aResultPresenter
     aResultPresenter result isFailed
             ifTrue:  [ Dialog warn:  'Nothing found' ]
             ifFalse:  [ SOULFindResultPresenter openOnResults:  aResultPresenter ]
```

This method of class `SOULFinderApp` takes an argument of type `SOULResultPresenter` and creates an instance of the view class `SOULFindResultPresenter`. The method `openOnResults:` indeed belongs to the instance-creation protocol of `SOULFindResultPresenter`. The rule which captures this abstract *code pattern* is given below:

```
% Example of code pattern 1:
% createsWith_C_C_C(SOULFinderApp, SOULResultPresenter, SOULFindResultPresenter)
createsWith_C_C_C(SourceClass, Type, ViewClass) :-
    % Is a direct Message sent from SourceClass to ViewClass?
    className(SourceClass, SrcClName),
    className(ViewClass, ViewName),
    isSentTo(SrcClName, SrcMthName, variable(ViewName), Message, Args),
    % Does the Message have an argument of the given Type?
    classImplementsMethodNamed(SourceClass, SrcMthName, Method),
```

```
    member(Arg, Args),
    mayHaveType_E_M_C(Arg, Method, Type),
    % Does the Message correspond to an instance-creation method of the ViewClass?
    metaClass(ViewClass, Meta),
    understands(Meta, Message),
    creationProtocolName(ProtocolName),
    methodNameInProtocolFlattened(Meta, methodProtocol(ProtocolName, _), Message).
% instance-creation protocols are:
creationProtocolName('instance creation').
creationProtocolName('view creation').
```

The above rule is defined in terms of the auxiliary predicates `isSentTo` and `mayHaveType_E_M_C`. We repeat that the `isSentTo` predicate performs a *parse-tree traversal* in search for some message send (see 6.2.4) and that the `mayHaveType_E_M_C` predicate tries to infer the type of some expression in the body of some method.

The above rule codifies the first way in which some `SourceClass` can create a `ViewClass` to view an instance of some class `Type`. The second way captures the more complex interaction where the `SourceClass` implements a method that sends a parameterless message (e.g., `inspect`) to an instance of class `Type`, which in turn sends a creation message to the `ViewClass` with itself as argument. As an example of this *interaction pattern*, consider the following Smalltalk method, which belongs to the class `SOULQueryApp`:

```
evaluate
     self evaluateEditField inspect
```

The method `evaluate` sends an argumentless message `inspect` to the result of the expression `self evaluateEditField`. The method `evaluateEditField` in this expression invokes a method `basicEvaluateEditField` which returns the result of interpreting a query. In other words, the expression `self evaulateEditField` is of type `SOULResultPresenter`. This class `SOULResultPresenter` indeed understands a method 'inspect'. Its implementation is shown below. It sends an instance-creation message `openOn:` to `SOULResultPresenterInspector` (the view class) with the current `SOULResultPresenter` (self) as argument.

```
inspect
     InputState default shiftDown
             ifTrue:  [ super inspect ]
             ifFalse:  [ SOULResultPresenterInspector openOn:  self ]
```

This complex interaction pattern is codified in the following Prolog rule:

```
% Example of code pattern 2:
% createsWith_C_C_C(SOULQueryApp, SOULResultPresenter, SOULResultPresenterInspector)
createsWith_C_C_C(SourceClass, Type, ViewClass) :-
    % Does the SourceClass send an argumentless Message to some Receiver?
    className(SourceClass, SrcClName),
    isSentTo(SrcClName, Method, Receiver, Message, []),
    % Is the Receiver of the expected Type?
    mayHaveType_E_M_C(Receiver, Method, Type),
    % Does the Method corresponding to the argumentless Message invoke a method
    % on the ViewClass?
    isSentTo(_ResultClass, Message, variable(TrgtClName), CreationMsg, Args),
    className(ViewClass, TrgtClName),
    % Is the invoked method an instance-creation method?
    metaClass(ViewClass, Meta),
    creationProtocolName(ProtocolName),
    methodNameInProtocolFlattened(Meta, methodProtocol(ProtocolName, _), CreationMsg),
    % Is 'self' passed as an argument?
    member(variable(self), Args).
```

As opposed to the first code pattern, this pattern is a non-local one: it involves multiple methods in multiple classes.

Virtual dependencies are often *defined in terms of other virtual dependencies.* As an example, consider the *isCreatedBy_C_C* predicate, which implements the **Is Created By** architectural relation. It is defined in terms of the more primitive `isPartOf_M_C` and `createsInstanceOf_M_C` virtual dependencies. The former verifies whether a method is part of some class; the latter checks whether some method creates an instance of some class.

```
% is an instance of Class1 created by (an instance of) Class2?
isCreatedBy_C_C(Class1, Class2) :-
    isPartOf_M_C(Method, Class2),
    createsInstanceOf_M_C(Method, Class1).
```

`createsInstanceOf_M_C` checks for the typical *coding patterns* that indicate the creation of an instance of a class. In Smalltalk, this is typically done by sending an instance-creation message. (The implementation of the predicate `instanceCreationMethod` is given in Appendix B.)

```
% Does method M create an instance of class C?
createsInstanceOf_M_C(M, C) :-
    % Does method M send a Message to class C?
    classImplementsMethodNamed(Class, MethodName, M),
    className(Class, ClassName),
    className(C, Receiver),
    isSentTo(ClassName, MethodName, variable(Receiver), Message),
    % Is Message the name of an instance-creation method?
    instanceCreationMethod(C, M),
    methodName(M, Message).
```

The above examples clearly illustrate the expressiveness of a LMP approach to declare virtual dependencies. The powerful techniques of parse-tree traversing, string pattern matching, unification and backtracking are used to codify complex coding patterns and interaction protocols. In addition we showed how virtual dependencies can be defined in terms of more primitive virtual dependencies.

7.1.6 Quantifiers

As explained in Subsection 5.3.1, quantifiers specify how a virtual dependency, associated with an architectural relation, should be abstracted to a relationship among architectural concepts. Since a virtual dependency only defines a relationship over single implementation artifacts, but architectural concepts are mapped to sets (i.e., virtual classifications) of such artifacts, a virtual dependency needs to be abstracted to a relationship among virtual classifications. The quantifiers specify how to consider the different artifacts in a virtual classification. For example, an $\exists$ quantifier means that it is sufficient for the intended relationship to hold for at least one of the artifacts in a virtual classification. The $\forall$ quantifier states that it should hold for all elements in a virtual classification.

The different quantifiers for the 'user interaction' view of Figure 7.1 should be interpreted as follows:

- *every* auxiliary application is created *either* by *some* type of user application *or* by some type of auxiliary application;
- *every* type of user application asks the repository to execute *some* request;
- *every* type of user application creates *some* kind of output window using *some* type of query result;
- *every* type of user application asks *some* part of the rule interpreter to interpret a query;

- *some* input window events trigger an action to activate *some* request on a user application *or* an auxiliary application.

Although other quantifiers than ∀ and ∃ are imaginable, in the case study we conducted we never felt the need to use any other quantifiers. We were able to express everything we wanted to in terms of these two quantifiers, in combination with well-chosen definitions of virtual classifications, virtual dependencies and port filters. When the need for other quantifiers would arise, however, they can readily be included in the architectural formalism.

7.1.7 Encountered difficulties

Problems with naming conventions

Overall, we were quite lucky with the naming conventions adopted by the SOUL developers. Many good naming conventions for both classes and methods were consistently used throughout the implementation. Also, the use of Smalltalk method protocols and class categories as built-in classification mechanisms provided by the Smalltalk language were often of great help in defining our virtual classifications.

However, using naming conventions to define virtual classifications does not always work. In Subsection 7.1.3, the 'resultViewer' virtual classification was defined as an enumeration of two separate class hierarchies (i.e., the hierarchies with root classes `SOULFindResultPresenter` and `SOULResultPresenterInspector`). Using naming conventions to define these classes did not work here, because the class names were not so well chosen. For example, although both classes `SOULFindResultPresenter` and `SOULResultPresenterInspector` contain the same string 'ResultPresenter' in their name, there are some other classes in the SOUL system that also contain this string but do not represent a result viewer class.

Apart from the fact that good naming conventions are not always available, on page 70 we mentioned some other problems with defining virtual classifications in terms of such naming conventions. Of course, we are not obliged to describe a virtual classification in terms of naming and coding conventions. In cases where it is not possible, or not opportune because the conventions are not consistently followed throughout the implementation, we can still define the virtual classification in other ways. First of all, we can use a classification which explicitly enumerates its elements (but then, of course, it would no longer be 'virtual'). Or, we can try to give a definition that is not based on conventions but that makes us of semantic inferencing. Such a more semantic definition is often harder to write, but is typically more robust towards changes.

For example, a more semantic definition of the 'resultViewer' virtual classification could be that 'resultViewer' classes are all classes that have an (instance or view) creation method which can take an argument of type `SOULResultPresenter`. This corresponds to the intuition that the result of a query, wrapped as an instance of the `SOULResultPresenter` class, is used by a result viewer class to create a view in which the result can be shown in an appropriate format to the user. Unfortunately, such a semantic definition is harder to write down than an extensional one, or one based on simple naming conventions. This particular intentional definition, for example, is not so easy to implement due to the lack of static type information in Smalltalk.

Lack of dynamic information

As already mentioned earlier, our current implementation takes a *static* approach towards architectural conformance checking. In most cases, this did not cause many problems, but sometimes we had difficulties expressing the things we wanted due to a lack of run-time information. For example, dynamic information might have helped in providing a more precise answer regarding the possible type of some expression.

We also encountered another example of a problem caused by a lack of dynamic information. In SOUL, when a user application invokes the query interpreter, a result is returned to the user application so that it can present *this* result to the user using some output viewer. Currently, the

'user interaction' view only captures this interaction partially. It does contain an architectural relation describing that the user application asks the query interpreter for some (query) result, as well as an architectural relation describing that the user application creates an output viewer with some query result. It does not describe however, that both query results should actually be the same object.[2] With our static approach, we can only reason about classes, methods and variables. To reason about objects, more dynamic information is needed.

It is important to note that our formalism, and the use of a LMP medium in which to reason about the software, does not inhibit a more dynamic approach. If we would have access to dynamic information, we could easily write logic predicates that reason about this information and define our virtual classifications and virtual dependencies in terms of these predicates. In fact, we consider this as an important area of future research.

Intentionality versus efficiency

Another difficulty we often had to face has to do with the trade-off between intentionality and efficiency. On the one hand, an intentional definition of a virtual classification or virtual dependency has the advantage of providing a more correct (more semantic) description, and is often more robust towards change. On the other hand, it is typically less efficient than a more pragmatic definition that is based, for example, on simple naming conventions. As a concrete example, reconsider the second rule defining which methods belong to the `'repository'` virtual classification:

```
classifiedAs(method('repository'), Method) :-
    findMethodsFromClasses(Method, 'soul'), % restrict scope to SOUL classes
    classifiedAs(class('repository'), Class),
    factoryMethod(Method, Class).
```

This rule uses the `factoryMethod` predicate to make sure that all 'factory methods' which create repositories are also included in the classification. However, we know that in the implementation of SOUL, all factory methods are localized in a single class `SOULFactory`. We also know that all factory methods for creating repositories are grouped in the method protocol 'repositories' on that class. Therefore, an alternative definition of the above rule could be to simply include all methods from that class and method protocol. The current definition is more intentional though, as it uses some kind of *semantic inferencing*, and is not based on the implicit convention that all these methods are grouped in the same protocol. Our intentional definition does not even mention the name of this class `SOULFactory`, nor the method protocol. On the downside, however, the intentional definition is rather time-consuming. (See Table 7.8 on page 137: computing the entire `'repository'` virtual classification takes more than two hours.)

In general, a more intentional definition should always be preferred whenever this is feasible. It is not necessarily a problem that a virtual classification takes a long time to compute. In our current implementation, every virtual classification is computed only once, after which its values are cached persistently so that they can be retrieved efficiently.

Parsing versus pattern matching

Finally, we discuss the trade-off between string pattern matching and parse-tree analysis. Both techniques can be used to define virtual dependencies and virtual classifications. Analyzing code by means of string pattern matching has the advantage of being very efficient. Analyzing a parse tree typically takes more time, but can provide more precise results. In our LMP language, we

[2] The problem is that by using two *independent* architectural relations ($Asks_1$ and *Creates With*), we loose important information that allows us to infer that the query result returned by $Asks_1$ is the same as the one that is used by *Creates With*. Nevertheless, it would be possible to describe this complex interaction statically, by using *one single* architectural relation which codifies the following check: "the user application invokes the query interpreter which returns some query result that is subsequently used by the user application to create an output viewer with it". However, the two original relations are much more simple, understandable and reusable than this highly specific and complex relation.

implemented some powerful predicates both for doing parse-tree analysis and for doing complex string pattern matching. Which technique to choose depends on the particular situation. Furthermore, nothing prevents us from combining both techniques, thus achieving the best of both worlds.

In Subsection 6.2.2, we explained the virtual dependency `asks_C_M`. It was defined in terms of an auxiliary predicate `isUsedBy_E_M`, which verifies whether a certain expression is used inside the body of some method.[3] To implement this predicate, there are two obvious alternatives. The first alternative is to use efficient string pattern matching to search for the expression inside the method body. However, due to a lack of information on the actual structure of the method, this string-based approach sometimes leads to false positives. A more precise alternative is to traverse the parse tree of the method in search for the desired expression. Such a parse-tree search is typically less efficient than performing a string pattern match, though.

To obtain a maximum of efficiency, without loosing precision, we decided to combine both approaches in our implementation of the `isUsedBy_E_M` predicate. First, string pattern matching is used to find the expression in the method, and then (if necessary), a parse-tree search is done to check whether the found match is not a false positive. Also note that our string pattern matching does not work on a string representing the Smalltalk code of some method, but on a string representation of the parse tree of that method. (All methods are stored in that format in the repository.) This representation has some advantages. It is easier to recognize expressions lexically, as they are all tagged with their kind. For example, every return statement is of the form '`return(...)`'.

Without going into the details, below we show our Prolog implementation of `isUsedBy_E_M`. We repeat that the auxiliary predicate `findMethod` is a powerful predicate for performing string pattern matches.

```
isUsedBy_E_M(E, M) :-
    write(E) ~> S,  % Convert expression E to a string S
    ( findMethod(_, M, pattern([_,'return(',S,')',_]))
      ->    true;   % is expression used as return statement?
      findMethod(_, M, pattern([_,'send(',S,_]))
      ->    true;   % is expression used as receiver of a message
      findMethod(_, M, pattern([_,'send(',_,'[',_,S,'])',_]))
      ->    % possibly used as the argument of a message
            ( classImplementsMethodNamed(C, MN, M), className(C, CN),
              isSentTo(CN, MN, _Rcvr, _Msg, Arguments), % parse-tree search
              member(E, Arguments) );
      findMethod(_, M, pattern([_,'assign(',_,S,')',_]))
      ->    % possibly used as assignment value
            assignStatement(M, _Var, E)                % parse-tree search
      ).
```

7.1.8 Timings

To get an idea of the time needed for checking conformance of the SOUL implementation to the 'user interaction' view, Table 7.7 lists the time needed to check each of the architectural relations, as well as the total time for checking the entire architectural view. The times were taken on a Pentium 133 processor with 16 MB of RAM; all other timings in this chapter were taken on the same machine.

We can conclude that for this particular architectural view, the conformance checking algorithm performs reasonably efficient (total time of about 16 minutes). However, since all virtual classifications were cached before conformance was checked, the above timings do not include the time needed for computing the virtual classifications associated with the different concepts in the

[3] Note that this predicate again illustrates the problems caused by a lack of dynamic information. Without dynamic information or extensive data-flow analysis, we cannot always know whether the expression will *actually* be used inside the method body. Therefore, we can only give an approximate answer.

Relation	Time (in seconds)
Activates	112
Is Created By	198
$Asks_1$	74
$Asks_2$	395
Creates With	191
Total	970

Table 7.7: Timings for checking conformance to the 'user interaction' view.

'user interaction' view. These timings are listed in Table 7.8. To give an idea of the size of the computed classifications, we also list the number of artifacts in each virtual classification.

Concept	Time (in seconds)	Number of classified artifacts
User Application	16	116
Output Viewer	23	66
Auxiliary Application	33	161
Input Window	13	51
Query Result	36	200
Repository	8640	268
Query Interpreter	343	162
Total	9104	

Table 7.8: Timings for computing the virtual classifications of the 'user interaction' view.

Most virtual classifications can be computed rather efficiently. Only for the **Repository** architectural concept, it took a long time to compute its classification (about 2.5 hours). This is because it was not based on simple naming conventions but had an intentional definition in terms of a virtual dependency. Also, it contains more artifacts than all other classifications. However, it is not such a problem that some virtual classifications may take a while to compute, as every virtual classification is computed only once and is then cached for further usage.

7.2 The rule-based interpreter architectural view

Now that we have explained in detail the 'user interaction' architectural view, we take a closer look at the 'rule-based interpreter' architectural view. For easy reference, we copied Figure 4.5 to Figure 7.2. We did not bother to annotate the links with quantifiers, as all links have the same quantifier ∃ attached.

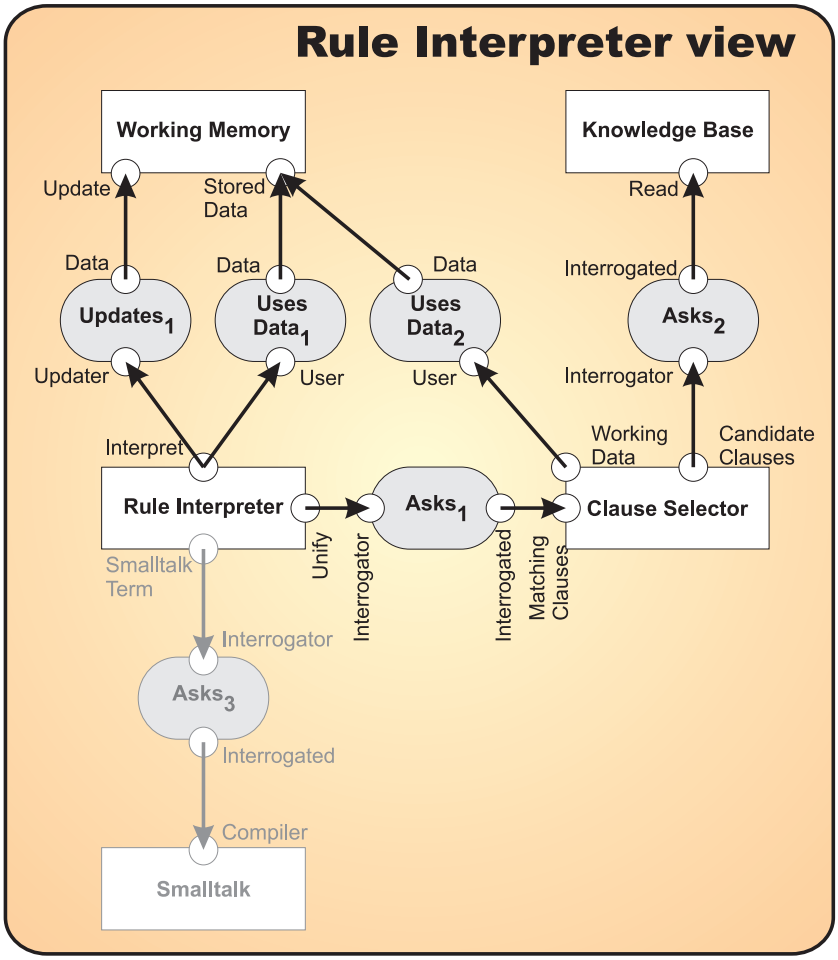

Figure 7.2: The 'rule-based interpreter' architectural view.

We will not show the facts that describe the architecture description and architectural instantiation for this particular architectural view. This is completely similar as for the 'user interaction' view (see Subsections 7.1.1 and 7.1.2). Instead, we focus on the definition of the architectural abstractions.

7.2.1 Virtual classifications

We start by explaining for each of the architectural concepts in the 'rule-based interpreter' view, to which virtual classification it is mapped (the architectural instantiation), and how this virtual classification is defined (the architectural abstraction).

The **Rule Interpreter** concept in the 'rule-based interpreter' view and the **Query Interpreter** concept in the 'user interaction' view are defined in terms of exactly the same virtual classification 'queryInterpreter'. They are simply two different views on the same thing (only their name and associated ports differ). This example motivates why it is useful to split up the

AML into two parts: an architectural instantiation language, which is used to associate virtual classifications with architectural concepts and an architectural abstraction language, which is used to define the virtual classifications. Indeed, here we have an example where the same virtual classification is used to instantiate two different architectural concepts in two different architectural views.

Similarly, both the **Knowledge Base** concept in the 'rule-based interpreter' view and the **Repository** concept in the 'user interaction' view are defined by exactly the same virtual classification 'repository'.

The virtual classification for **Working Memory** is an example of a virtual classification which simply enumerates the classes that belong to it. It contains both the class `SOULBindings` and the class `SOULResult` (and their subclasses). Intuitively, the **Working Memory** remembers the bindings of logic variables to values that were made during the logic interpretation process. `SOULBindings` represents a kind of dictionary, associating logic variables with their values. And the class `SOULResult` not only represents the result of interpreting a SOUL clause, but is also used to pass around bindings. It contains the necessary methods for updating (i.e., adding, removing or inspecting) the current bindings.

```
classifiedAs(class('workingMemory'), Class) :-
    hierarchy(class('SOULBindings',_), Class);
    hierarchy(class('SOULResult',_), Class).

% 'workingMemory' methods are methods that belong to 'workingMemory' classes.
classifiedAs(method('workingMemory'), Method) :-
    findMethodsFromClasses(Method, 'workingMemory').
```

When looking at the implementation of SOUL, the **Clause Selector** architectural concept seems to correspond to a Smalltalk method `calculateUnifiers: aRepository` which tries to calculate all clauses, in some knowledge base `aRepository`, that unify with the current clause under investigation. There are many of these methods: one for each kind of clause in the SOUL abstract grammar.

Unfortunately, things are not quite that simple. At closer inspection, the `calculateUnifiers:` methods do not seem to directly access the knowledge base, as is required by the architecture. This is because the `calculateUnifiers:` methods are only very small methods which delegate most of their work to other methods, which in turn may delegate part of their work to yet other methods. It are some of these indirectly called methods which eventually access the knowledge base. For example, the method `calculateUnifiers: aRepository` on the Smalltalk class `SOULNamedTerm` invokes an auxiliary method `unifyingClauses: aRepository` which in turn calls a method `candidateClauses: aRepository` which eventually asks the knowledge base to return all clauses matching the current term.

```
calculateUnifiers: aRepository
    ...
    (self unifyingClauses: aRepository) do: ...
    ...
unifyingClauses: aRepository
    ...
    (self candidateClauses: aRepository) do: ...
    ...
candidateClauses: aRepository
    ^aRepository clausesForTerm: self
```

It is important to note that the parameter `aRepository` representing the knowledge base that is eventually accessed, is passed around through all these invocations. This *pattern of collaboration* occurs frequently in object-oriented implementations: one method delegates much of its work to others while passing them the necessary parameters.

To capture this intuition that the **Clause Selector** corresponds to all `calculateUnifiers: aRepository` methods as well as all their 'auxiliary' methods, we define it in terms of a virtual

classification which computes the *reflexive and transitive closure* of all methods invoked by a `calculateUnifiers: aRepository` method *and* with at least *one argument in common*. Since computing a transitive closure can be very computationally intensive, we reduce its scope by considering only methods that belong to classes representing clauses in the SOUL abstract grammar.

```
classifiedAs(method('ruleSelection'), Method) :-
    closure( invokesWithSameArgumentInSOULSyntax,
             method('calculateUnifiers:',_),
             Method ).
```

The above predicate uses the second-order predicate `closure(Relation, Start, Target)` which computes every `Target` in the reflexive and transitive closure of some binary `Relation`, for some starting point `Start`. The particular relation `invokesWithSameArgumentInSOULSyntax` we are interested in, is the method invocation relationship, but with the extra restriction that one of the caller's arguments is passed with the invocation, and with the scope restricted to 'soulSyntax' methods. 'soulSyntax' is an *auxiliary virtual classification*, which was specifically defined to restrict the scope of classes and methods in the 'ruleSelection' classification. It is defined as the set of all classes representing clauses or terms in the SOUL abstract grammar, together with their methods.

```
classifiedAs(class('soulSyntax'), Class) :-
    hierarchy(class('SOULAbstractTerm',_), Class);
    hierarchy(class('SOULClause',_), Class).

% 'soulSyntax' methods are methods that belong to 'soulSyntax' classes.
classifiedAs(method('soulSyntax'), Method) :-
    findMethodsFromClasses(Method, 'soulSyntax').
```

The main difference between SOUL and Prolog is SOUL's close symbiosis with Smalltalk. More specifically, SOUL provides a special kind of 'Smalltalk terms' that allow for Smalltalk blocks, as part of logic clauses, to be computed during the interpretation process. Interpreting 'Smalltalk terms' is achieved by directly calling the **Smalltalk** system. The classes belonging to the core of Smalltalk can be recognized easily, as they all belong to one of the 'System' categories ('System-Compiler Support', 'System-Code Storage', ...). All these categories start with the same string 'System-'.

```
classifiedAs(class('smalltalk'), Class) :-
    categoryName(Category, SystemCategory),
    stringStartsWith(SystemCategory, 'System-'),
    classInCategory(Category, Class).
```

Again, the defined virtual classifications were concise, intuitive and readable. Some were straightforward, merely enumerating some class hierarchies, or making use of simple naming conventions or tagging information. Others required some more intricate semantic inferencing: to define the virtual classification 'ruleSelection' we had to compute the reflexive and transitive closure of the invocation relationship, with the extra constraint that one argument should remain the same over the transitive invocations.

We also illustrated that there is not always a one-to-one mapping between virtual classifications and architectural concepts. Some virtual classifications, e.g., the 'soulSyntax' classification, may not instantiate any architectural concept, but serve only as an auxiliary building block for defining other virtual classifications. Other virtual classifications, e.g., 'queryInterpreter' or 'repository', are used to instantiate multiple concepts in different architectural views.

7.2.2 Port filters

As in the previous architectural view, most ports in the 'rule-based interpreter' view are mapped to the *predefined class and method filters* of the DFW. In fact, most ports represent actions (i.e., reading, interpreting, unifying, updating, matching clauses, selecting candidate clauses and accessing working data) and are therefore mapped to a method filter. Only the port 'Stored Data', representing the kind of data stored in the **Working Memory**, is mapped to a class filter.

```
portMapping(soulRuleBasedSystem, knowledgeBase,   read,             methodFilter).
portMapping(soulRuleBasedSystem, clauseSelector,  matchingClauses,  methodFilter).
portMapping(soulRuleBasedSystem, clauseSelector,  candidateClauses, methodFilter).
portMapping(soulRuleBasedSystem, clauseSelector,  workingData,      methodFilter).
portMapping(soulRuleBasedSystem, ruleInterpreter, interpret,        methodFilter).
portMapping(soulRuleBasedSystem, ruleInterpreter, unify,            methodFilter).
portMapping(soulRuleBasedSystem, workingMemory,   update,           methodFilter).
portMapping(soulRuleBasedSystem, workingMemory,   storedData,       baseClassFilter).
```

In addition to these predefined port filters, ports can be mapped to *user-defined filters* as well.

```
portMapping(soulRuleBasedSystem, ruleInterpreter, smalltalkTerm, smalltalktermClassFilter).
portMapping(soulRuleBasedSystem, smalltalk, compiler,            compilerClassFilter).
```

As an example, consider the port 'Smalltalk Term' of the **Rule Interpreter** concept. The classification associated with **Rule Interpreter** contains (amongst others) all classes implementing interpretation methods, that is, practically every class which represents a node of the SOUL abstract grammar. Mapping the 'Smalltalk Term' port to the predefined 'class filter' would yield all these classes. This does not correctly capture our intention that the port corresponds to Smalltalk terms only. Therefore, we use a filter which takes only a subset of all classified classes, namely the class `SOULAdvancedSmalltalkTerm` and its subclasses, which represent precisely the Smalltalk terms.

```
smalltalktermClassFilter(Artifact) :-
    classFilter(Artifact),
    hierarchy(class(SOULAdvancedSmalltalkTerm,_), Artifact).
```

The attentive reader may have noticed that, in our explanation of the 'smalltalk' virtual classification in the previous subsection, we did not specify precisely which part of the **Smalltalk** system is accessed by the Rule Interpreter, when Smalltalk terms need to be computed. The actual computation of Smalltalk terms will be done by the Smalltalk compiler. Therefore, the port 'Compiler' will restrict the set of all classes representing the **Smalltalk** system, to the 'Compiler' classes only.

```
compilerClassFilter(Artifact) :-
    classFilter(Artifact),
    % Does the class Artifact contain the string 'Compiler' in its name?
    className(Artifact, Name),
    stringContains(Name, 'Compiler').
```

Instead of using the above user-defined filter, we could have included the filtering of all classes that contain a string 'Compiler' directly in the definition of the 'smalltalk' virtual classification. We will explain in Subsection 7.2.5 why we preferred not to do so.

The above examples illustrate that it is both useful and possible (and easy) to instantiate ports with user-defined filters. Furthermore, these filters can themselves be defined in terms of more primitive filters. For example, both the user-defined filters `smalltalktermClassFilter` *and* `compilerClassFilter` *use the predefined filter* `classFilter`*.*

7.2.3 Virtual dependencies

Both the $Asks_1$ and $Asks_2$ relations of the 'rule-based interpreter' architectural view are linked at both ends to ports that are mapped to method filters. Therefore, we map both architectural relations to the `asks_M_M` virtual dependency between methods, which is defined in terms of its symmetric predicate `isAskedBy_M_M`. This predicate `isAskedBy_M_M` was already introduced and explained in Subsection 6.2.4, as an auxiliary predicate for the definition of `asks_C_M`.

```
asks_M_M(M1, M2) :-
    isAskedBy_M_M(M2, M1).
```

The $Asks_3$ architectural relation between **Rule Interpreter** and **Smalltalk** codifies the knowledge that the rule interpreter requests the Smalltalk system to compile a Smalltalk term. Because both the ports 'Smalltalk Term' (on **Rule Interpreter**) and 'Compiler' (on **Smalltalk**) generate classes, we define the $Asks_3$ architectural relation in terms of the `asks_C_C` virtual dependency between classes. `asks_C_C` itself is defined directly in terms of `asks_C_M`:

```
asks_C_C(C1, C2) :-
    classImplementsMethod(C2, M2),
    asks_C_M(C1, M2).
```

A *Uses Data* architectural relation expresses the fact that some architectural concept uses some kind of data provided by another architectural concept. In the case of the architectural relations *Uses* $Data_1$ and *Uses* $Data_2$, the used data corresponds to one of the **Working Memory** classes. In the *Uses* $Data_1$ relation, data is used by the interpretation process; in the *Uses* $Data_2$ relation, it is accessed during the selection and matching of clauses. When looking at the implementation of SOUL, the *Used Data* architectural relation boils down to the fact that interpretation methods or clause selection/matching methods take some instance of a Working Memory class as argument. Therefore, we implement both *Uses* $Data_1$ and *Uses* $Data_2$ in terms of the virtual dependency `hasParameterType_M_C` which checks whether some method has a parameter of some type (class).

```
hasParameterType_M_C(M, C) :-
    methodArgument(M, Argument),
    argumentVarName(Argument, VarName),
    mayHaveType_V_M_C(VarName, M, C).
```

Finally, the *Updates* architectural relation can be checked straightforwardly by verifying whether some mutator method is invoked directly or indirectly. A mutator method is a method that updates the value of some variable (see Appendix B). An indirect mutator is one that does not perform the update itself, but transitively invokes a direct mutator, passing it the new value for the variable.

```
% invokesMutator_M_M checks whether a method M1 invokes a mutator method M2
invokesMutator_M_M(M1, M2) :-
    % Is M2 a direct or indirect mutator method?
    ( mutatorMethod(M2); indirectMutatorMethod(M2) ),
    % Does M1 invoke M2 ?
    invokes_M_M(M1, M2).
```

Similar to the `createsWith_C_C_C` virtual dependency, the `invokes_M_M` virtual dependency in the predicate above is defined in terms of the auxiliary predicates `isSentTo` and `mayHaveType_E_M_C`.

```
invokes_M_M(M1, M2) :-
    classImplementsMethodNamed(Class, MethodName, M1),
    className(Class, ClassName),
    methodName(M2, Message),
    isSentTo(ClassName, MethodName, Receiver, Message),
    mayHaveType_E_M_C(Receiver, M1, ReceiverClass),
    classImplementsMethod(ReceiverClass, M2).
```

7.2.4 Quantifiers

The architectural relations in the 'rule-based interpreter' architectural view explain how *some* parts of the rule interpretation process interact with *some* parts of the working memory, knowledge base, Smalltalk system, and clause selector; as well as how *some* parts of the clause selection process interact with *some* parts of the working memory and knowledge base. Therefore, it sufficed to add a simple ∃ quantifier to all links in this architectural view. Below, we show some of the link mappings which associate this quantifier to the links.

```
linkMapping(soulRuleBasedSystem, clauseSelector, candidateClauses, asks2, interrogator,
            exists).
linkMapping(soulRuleBasedSystem, knowledgeBase, read, asks2, interrogated, exists).
linkMapping(soulRuleBasedSystem, clauseSelector, workingData, usesData2, user, exists).
linkMapping(soulRuleBasedSystem, workingMemory, storedData, usesData2, data, exists).
...
```

7.2.5 Encountered difficulties

Predefined versus user-defined port filters

In Subsection 7.2.2, we encountered some examples of user-defined port filters. E.g., the 'Compiler' port of the **Smalltalk** concept was defined in terms of a user-defined filter `compilerClassFilter`. An alternative solution would have been to define the **Smalltalk** concept in terms of a virtual classification that contains only the compiler classes in the Smalltalk system. In that case, we could have defined the 'Compiler' port in terms of a predefined `baseClassFilter`. However, this alternative way of modeling things does not correctly capture the intention of the **Smalltalk** concept, which intuitively corresponds to the core of the Smalltalk system, and not only to the compiler classes. This would lead to problems when the **Smalltalk** concept would participate in other architectural relations.

Generic virtual dependencies

We already encountered several variants of the `asks` virtual dependency: one for classes, one for methods, and so on. The variant that is chosen to instantiate a particular architectural relation depends on the kind of artifacts that are generated by the ports to which this relation is linked. This implies that, when the port filters are changed, the virtual dependency may need to be changed as well. But in fact, there is no reason why we cannot define a more general 'asks' virtual dependency, that is more robust towards such changes, and that works for all kinds of artifacts. Depending on the kind of artifacts it is called with, it simply decides which more specific variant of the asks virtual dependency to call. Of course, the same reasoning holds for other virtual dependencies as well.

The predicate below implements such a more generic `asks` virtual dependency. It contains a big case statement (the notation with '`->`' and '`;`' is Prolog syntactic sugar for a case statement) which chooses the appropriate variant depending on the types of the provided arguments. Instead of implementing such a generic version for every virtual dependency, a cleaner solution would be to use the technique of 'generic operations' [1] where a translation matrix decides how to translate a certain generic operation into a more specific one, based on the types of the arguments provided.[4]

```
asks(A1, A2) :-
    (class(A1), class(A2))   -> asks_C_C(A1, A2);
    (class(A1), method(A2))  -> asks_C_M(A1, A2);
    (method(A1), method(A2)) -> asks_M_M(A1, A2);
    ...
```

[4] This is the approach we took in [52].

7.2.6 Timings

Table 7.9 lists the timings for checking the architectural relations in the 'rule-based interpreter' view. (Note that the times are indicated in minutes, not in seconds.) All relations could be checked in a reasonable time. Only the $Asks_2$ relation took a very long time (more than 24 hours).

Relation	Time (in minutes)
$UsesData_1$	6
$UsesData_2$	1
$Updates_1$	210
$Asks_1$	31
$Asks_2$	(> 24 hours)
$Asks_3$	26

Table 7.9: Timings for checking conformance to the 'rule-based interpreter' view.

As before, in Table 7.10 we mention the time needed for computing the virtual classifications associated with the different concepts in this architectural view, as well as the number of classified artifacts for each of these virtual classifications. The timing for the **Knowledge Base** concept is exactly the same as that for **Repository** concept in Table 7.8, as they are both mapped to the same classification. The same holds for the architectural concept **Rule Interpreter**, which is mapped to the same classification as the concept **Query Interpreter** in the 'user interaction' view. These timings were not included in the total: if we assume that the virtual classifications for the 'user interaction' view have already been computed, the virtual classifications for **Knowledge Base** and **Rule Interpreter** have been cached and will not be re-computed.

Concept	Time (in seconds)	Number of classified artifacts
Working Memory	23	106
Clause Selector	456	48
Smalltalk	37	258
Knowledge Base	8640	268
Rule Interpreter	343	162
Total	516	

Table 7.10: Timings for computing the virtual classifications of the 'rule-based interpreter' view.

We can conclude from Table 7.10 that the time needed for computing the virtual classifications is acceptable.

7.3 The application architecture view

The architectural views discussed in the previous two sections both focused on a specific concern of the system (i.e., 'rule-based interpretation' and 'user interaction'). These architectural views express the important concepts and relations for those particular concerns, independent of how the software itself is structured. As a consequence, the architectural mapping for those views cuts across the implementation structure. In this section, we discuss the 'application architecture' of SOUL, which does explicitly focus on the implementation structure. It identifies the main implementation components of the system as well as how they relate to each other.

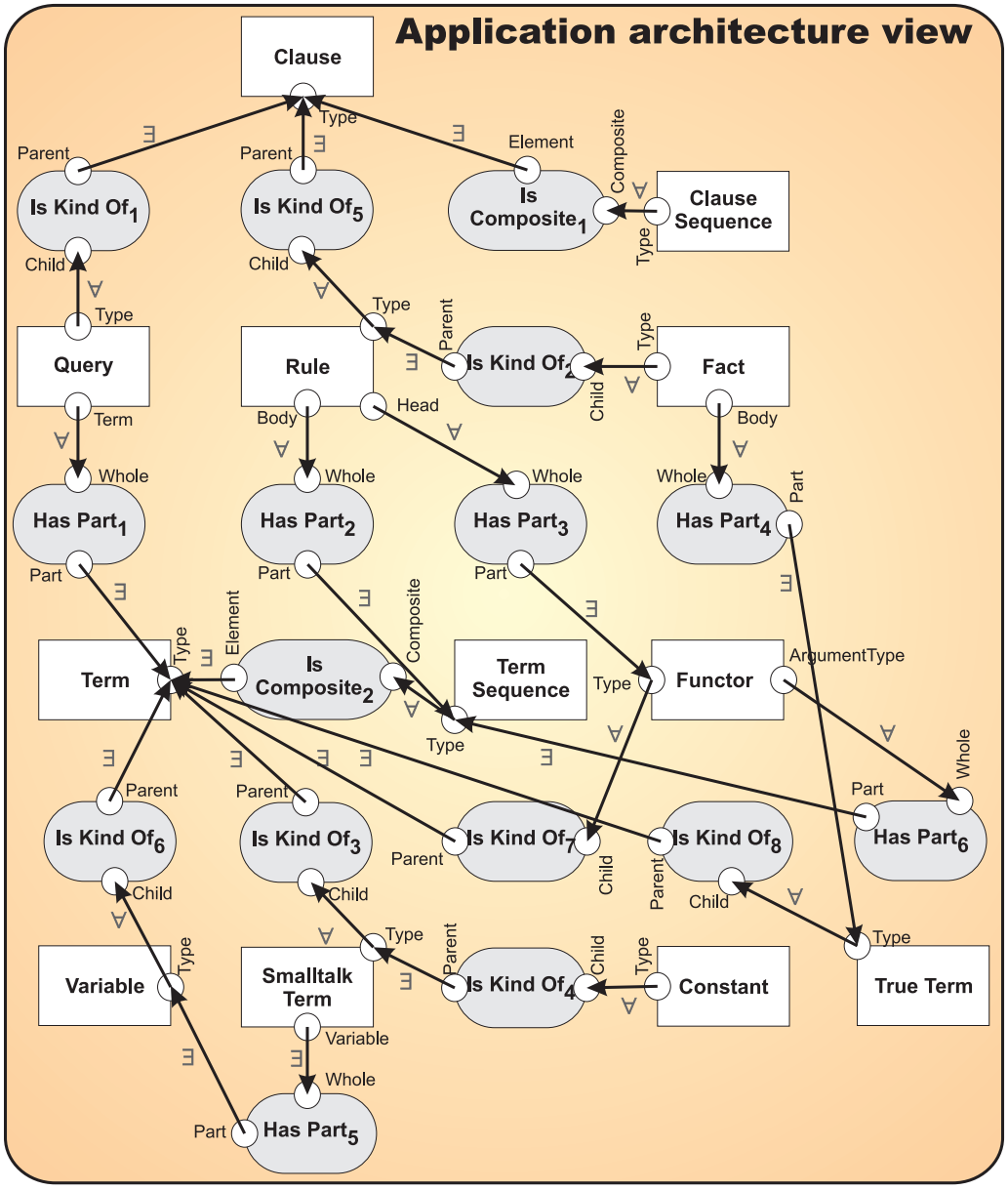

Figure 7.3: The 'application architecture' view with quantifiers.

In Figure 7.3, we repeat the entire 'application architecture' view of Figure 4.6, with a quantifier added to every link.

7.3.1 Virtual classifications

The 'application architecture' view describes the general implementation structure of the SOUL system. The concepts of interest in this architectural view are the important implementation components. For an object-oriented implementation, these components are typically classes or class hierarchies (i.e., an abstract class and its concretizing classes). Therefore, most virtual classifications corresponding to the different architectural concepts in the 'application architecture' view are defined in terms of the `hierarchy` predicate.

```
classifiedAs(class('clause'), Class) :-         % Clause
    hierarchy(class('SOULClause',_), Class).
classifiedAs(class('clauses'), Class) :-        % Clause Sequence
    hierarchy(class('SOULClauses',_), Class).
classifiedAs(class('query'), Class) :-          % Query
    hierarchy(class('SOULQuery',_), Class).
classifiedAs(class('rule'), Class) :-           % Rule
    hierarchy(class('SOULRule',_), Class).
classifiedAs(class('fact'), Class) :-           % Fact
    hierarchy(class('SOULFact',_), Class).
classifiedAs(class('term'), Class) :-           % Term
    hierarchy(class('SOULAbstractTerm',_), Class).
classifiedAs(class('terms'), Class) :-          % Term Sequence
    hierarchy(class('SOULTerms',_), Class).
classifiedAs(class('compoundTerm'), Class) :-   % Functor
    hierarchy(class('SOULCompoundTerm',_), Class).
classifiedAs(class('variable'), Class) :-       % Variable
    hierarchy(class('SOULVariableTerm',_), Class).
classifiedAs(class('smalltalkTerm'), Class) :-  % Smalltalk Term
    hierarchy(class('SOULSmalltalkConstantTerm',_), Class).
classifiedAs(class('trueTerm'), Class) :-       % True Term
    hierarchy(class('SOULTrueTerm',_), Class).
```

Only the virtual classification 'constant' is not defined as a class hierarchy, but as a single class `SOULSmalltalkConstantTerm`. This is because it contains some subclasses which do not present constants, but 'Smalltalk terms'.

```
classifiedAs(class('constant'), Class) :-       % Constant
    className(Class, 'SOULSmalltalkConstantTerm').
```

In fact, in the SOUL implementation, constant terms themselves are a special kind of 'Smalltalk terms' that do not contain any variables. The subclasses of `SOULSmalltalkConstantTerm` extend this functionality to include more complex kinds of 'Smalltalk terms'. This is why we defined the 'smalltalkTerm' virtual classification as the class hierarchy that has `SOULSmalltalkConstantTerm` as its root.

7.3.2 Port filters

Because all architectural concepts in this view are mapped to virtual classifications that are defined in terms of classes or class hierarchies, every port in the 'application architecture' view of SOUL is mapped to a class filter. Alternatively, we could have chosen to map them to identity filters, which would have exactly the same result (as the classifications contain nothing but classes anyway). However, if in the future we would decide to add other kinds of artifacts to one of the classifications, the class filter will still generate only classes, whereas the identity filter would not.

```
portMapping(soulApplication, clause, type,            baseClassFilter).
portMapping(soulApplication, clauseSequence, type,    baseClassFilter).
portMapping(soulApplication, query, type,             baseClassFilter).
portMapping(soulApplication, query, term,             baseClassFilter).
...
portMapping(soulApplication, smalltalkTerm, type,     baseClassFilter).
portMapping(soulApplication, smalltalkTerm, variable, baseClassFilter).
portMapping(soulApplication, constant, type,          baseClassFilter).
```

7.3.3 Virtual dependencies

Every *Is Kind Of* relation is mapped to a virtual dependency `specializes_C_C(C1, C2)`, which checks whether a class `C1` is a (direct or indirect) subclass of a class `C2`.

```
specializes_C_C(C1, C2) :-
    hierarchy(C2, C1).
```

Every *Has Part* architectural relation is mapped to a virtual dependency `hasPart_C_C`, which checks whether a class `Part` is a part of a class `Whole`. We do this by checking whether `Whole` has an instance variable of type `Part`. We make use of the auxiliary predicate `instVarTypes` which computes the possible types of some instance variable `Var` of the class `Whole`. (We will discuss this predicate in more detail later in subsection 7.3.5).

```
hasPart_C_C(Whole, Part) :-
    instVarTypes(Whole, Var, TypeList),
    member(Part, TypeList).
```

Every *Is Composite* architectural relation is mapped to a virtual dependency `isComposite_C_C`, which checks whether some `Composite` class is a special kind of class `Type` and is a container of elements of that `Type`. It can be defined in terms of the previously discussed virtual dependency `specializes_C_C` and a new virtual dependency `containsElementsOfType_C_C`. (We will discuss this new predicate in more detail later in subsection 7.3.5).

```
isComposite_C_C(Composite, Type) :-
    specializes_C_C(Composite, Type),
    containsElementsOfType_C_C(Composite, Type).
```

In the 'application architecture', some architectural relations can be mapped straightforwardly to simple implementation dependencies. For example, the Is Kind Of *architectural relation corresponds to the inheritance implementation relationship, and the* Has Part *architectural relation to variable containment. Other architectural relations in the 'application architecture' view, such as the* Is Composite *relation, did not directly correspond to implementation dependencies, but represented more complex relationships.*

7.3.4 Quantifiers

In this architectural view, all links (except for one) originating in an architectural concept have a $\forall$ quantifier attached, and every link arriving in an architectural relation has an $\exists$ quantifier attached. Again we repeat that the direction of links has no real semantics associated with it. It just aids an architect in reading the architecture: we adopt the convention that an incoming arrow on a relation is the 'subject' of the relation, and the outgoing arrows are 'direct and indirect objects'. For example, the *Is Kind Of*$_2$ relation should be interpreted as: every fact is some kind of rule; and the *Has Part*$_1$ relation as: every query has some kind of term as a part.

```
linkMapping(soulApplication, clauseSequence, type, isComposite1, composite, forall).
linkMapping(soulApplication, clause, type, isComposite1, element, exists).
linkMapping(soulApplication, query, type, isKindOf1, child, forall).
linkMapping(soulApplication, clause, type, isKindOf1, parent, exists).
linkMapping(soulApplication, query, term, hasPart1, whole, forall).
linkMapping(soulApplication, term, type, hasPart1, part, exists).
...
```

The only exception is the relation *Has Part*$_5$ where the originating link has an $\exists$ quantifier. This is because *not* every kind of **Smalltalk Term** can contain variables. In the implementation of SOUL, there are multiple classes representing 'Smalltalk terms' in the SOUL language: `SOULSmalltalkConstantTerm`, `SOULAdvancedSmalltalkTerm` and `SOULSmalltalkMetaPredicate`. The first `SOULSmalltalkConstantTerm` is used only to make Smalltalk constants available to SOUL, and cannot contain any variables. The other two represent more complex Smalltalk terms that can reference logic variables.

7.3.5 Encountered difficulties

Lack of type information

Again we encountered some difficulties due to the lack of type information in Smalltalk. For example, for the `hasPart_C_C` virtual dependency we wanted to determine whether some class `Whole` had some instance variable of type `Part`. To compute the type of this instance variable, we made use of an auxiliary predicate `instVarTypes`. Due to the lack of static type information, this predicate only yields an approximate answer. It infers the possible types of an instance variable in some class `Whole` by statically looking at all the messages sent to that variable (from the class `Whole` up to the first superclass that implements the variable). All classes that understand all these messages are then accumulated in a `TypeList`. Every one of these classes is a possible candidate for the type of the variable (it is in general impossible to find a unique type statically). After calling this auxiliary predicate `instVarTypes`, the only thing that remains to be done by `hasPart_C_C` is to check whether the class `Part` belongs to this list of possible types.

```
instVarTypes(Class, InstVar, TypeList) :-
    % does Class or one of its superclasses contain a variable InstVar?
    instVarFlattened(Class, InstVar),
    % compute the set of all Messages sent to this variable
    instVarName(InstVar, InstVarName),
    instVarMessages(Class, InstVarName, Messages),
    Messages \= [], % Fail if no messages are sent to the variable
    % compute all classes that understand all these Messages
    findall(Type, understandsAll(Type, Messages), TypeList).
```

Similarly, the virtual dependency `isComposite_C_C` was defined in terms of an auxiliary predicate `containsElementsOfType_C_C` which statically tries to derive the possible types of the elements contained in some instance variable representing a container of elements. Without going into all the technical details, the predicate proceeds as follows: the container variable is typically manipulated by iterating over its elements. During such an iteration, messages are sent to the contained elements. In a way similar to what is done in the predicate `instVarTypes`, from these message sends we can derive the type of the elements.

In both examples, the virtual dependency only provides an approximate answer, where the precision of the answer depends on the amount of messages that is sent to the instance variable of which we need to compute the type. The more information we have, the more precise the type can be guessed.

One could argue that problems such as these make it much harder to define virtual dependencies. This is not entirely true. In fact, our predicates for inferring the types of expressions should not be regarded as being part of the virtual dependencies, but belong to the base layer of

the DFW (see 5.3.5). They are predefined primitive predicates that are specifically designed for reasoning about Smalltalk code. When defining new virtual dependencies, we can make use of any of these predefined predicates, and do not need to bother with how they should be implemented.

7.3.6 Timings

Checking architectural conformance was rather efficient. Table 7.11 lists the time needed to check each of the architectural relations, as well as the total time needed for checking conformance of the SOUL implementation to the entire application architecture view.

Relation	Time (in seconds)
Is Kind Of_1	10
Is Kind Of_2	4
Is Kind Of_3	20
Is Kind Of_4	4
Is Kind Of_5	8
Is Kind Of_6	12
Is Kind Of_7	17
Is Kind Of_8	6
Is $Composite_1$	12
Is $Composite_2$	74
Has $Part_1$	30
Has $Part_2$	97
Has $Part_3$	97
Has $Part_4$	34
Has $Part_5$	76
Has $Part_6$	132
Total	633

Table 7.11: Timings for checking conformance to the 'application architecture' view.

Since all virtual classifications were cached before conformance was checked, the above timings do not include the time needed for computing the virtual classifications. Thanks to the straightforward mapping of the virtual classifications to classes and class hierarchies, and because they do not contain many elements, it does not take long to compute the virtual classifications of the 'application architecture' view. See Table 7.12.

Concept	Time (in seconds)	Number of classified items
Clause	15	7
Clause Sequence	6	1
Query	6	1
Rule	7	3
Fact	6	1
Term	9	18
Term Sequence	6	3
Functor	6	4
Smalltalk Term	3	4
True Term	6	1
Variable	6	2
Constant	4	1
Total	80	

Table 7.12: Timings for computing the virtual classifications of the 'application architecture' view.

7.4 Dealing with conformance conflicts

The Smalltalk implementation of SOUL was conform to all architectural views explained in the previous sections. However, during the extraction of an architectural view from the implementation, or when implementing a software system in accordance with an architecture view, we often encounter situations where the implementation is not in conformance with the architecture. We call such a situation an architectural *conformance conflict*. We provide an example of a conformance conflict, explain how it was resolved, and show how the conformance checking algorithm can be used to help in detecting the cause of such conflicts.

7.4.1 Example of a conformance conflict

As an example, we discuss a conformance conflict that we actually encountered during our experiments with the 'application architecture' view. As functors are expressions of the form $f(a_1, \ldots, a_n)$, where each argument a_i can either be bound or not, we originally modeled this at the architectural level by declaring a *Has Part* relation between **Functor** on the one hand, and **Constant** and **Variable** on the other hand, as depicted in Figure 7.4.

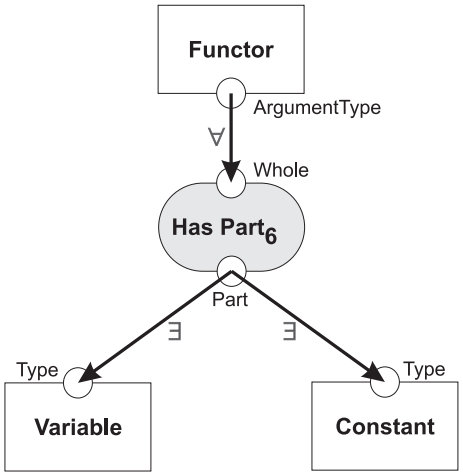

Figure 7.4: A non-conform architectural relation.

When checking conformance of the SOUL implementation to an 'application architecture' view containing the architectural relation of Figure 7.4, our conformance checking algorithm decided that the implementation did not satisfy this architectural view. After identifying the source of this failure (Subsection 7.4.2 explains how this was done), we realized that the problem was caused by the architectural relation *Has Part*$_6$. More specifically, the problem was that the class `SOULList` (which was classified as a **Functor**) did not have any of the classes representing a **Variable** (i.e., `SOULVariableTerm` or `SOULUnderscoreVariableTerm`) or a **Constant** (i.e., `SOULSmalltalkConstantTerm`) as its part. This contradicted the described relation that every class representing a functor could have a variable or a constant as its part.

To resolve the problem, we manually tried out the predicate `hasPart_C_C(Whole,Part)` with `Whole` bound to the problematic class `SOULList` and the second argument `Part` left uninstantiated. Thus, we found out that the only possible parts for class `SOULList` (i.e., the only results for `Part` generated by the query) were the class `SOULTerms` and it subclasses. At that point we realized that the algorithm probably made the correct decision: a functor should not directly have variables or constants as its parts, but an ordered collection of terms representing the argument list. In other words, a **Functor** has as its part a **Term Sequence**. When modeling the *Has Part*$_6$ relation like this (as we did in Figure 7.3), the conformance check did succeed.

This new way of representing the relation also solved another problem with the representation of Figure 7.4. In that representation, it seems as if a functor can have only constants or variables as arguments. This is not correct. Every kind of term, including Smalltalk terms and functors themselves, can be an argument of a functor. With the new representation, a functor indeed has an argument list which is a term sequence that can contain any kind of term. Yet another problem with the representation of Figure 7.4 is that it did not allow a functor to have an empty argument list.

A similar conformance checking conflict was encountered with the *Is Composite* architectural relations. Originally, the `isComposite_C_C` virtual dependency (to which the *Is Composite* architectural relations are mapped) was defined as a conjunction of the `specializes_C_C` and `hasPart_C_C` virtual dependencies. Again, this was not correct because a *Has Part* relation was not sufficient here. We should not check whether the composite contains a variable of a certain type, but whether it contains a *collection of elements* which are of a certain type.

7.4.2 Resolving conformance conflicts

Our original conformance checking algorithm is rather primitive, merely returning a 'true' or 'false' depending on the success of a conformance check. This information is insufficient: knowing only that the implementation of a software system does *not* conform to the architecture, does not provide enough information to fix the problem. We need more fine-grained information on which architectural relation is violated and on what are the implementation-level artifacts and dependencies that caused this violation.

For example, let us revisit the conformance conflict we discussed in Subsection 7.4.1. Our original conformance checking algorithm simply returned false when the conformance conflict was encountered. To obtain a better insight in the actual problem, more information was required. To start with, we needed to know which architectural relations caused the problem. In this particular example, it was the *Has Part*$_6$ relation between **Functor** and **Variable**, depicted in Figure 7.4. For this particular relation, and taking into account the quantifiers on its edges, we needed to know for which particular implementation artifacts conformance was invalid. In this case, non-conformance was caused by the class `SOULList` which did not have any of the classes `SOULVariableTerm`, `SOULUnderscoreVariableTerm` or `SOULSmalltalkConstantTerm` as its part. This information eventually allowed us to understand and resolve the conflict, as explained in Subsection 7.4.1.

Although the above information was not provided by our original implementation of the conformance checking algorithm, it was easy to extend its implementation to provide this information. The ease by which this could be done is mainly due to the power of LMP. We only had to use some 'more intelligent' quantifier predicates. For example, instead of using the primitive Prolog predicate `forall`, which checks some logic expression for some set of generated values, we used `forallDebugOne` (see Table 5.10 on page 78 and see page 111) which reports the first value that failed to satisfy the expression (if any). Upon failure of the conformance check, this reported information could be inspected to see which values caused the `forall` predicate (or more precisely, the `forallDebugOne` predicate) to fail.

Returning to the example, the first occurrence where a `forall` is used is to check conformance of all architectural relations. Upon failure, `forallDebugOne` reports which particular architectural relation fails (i.e., *Has Part*$_6$). Failure of checking conformance for this architectural relation occurs when checking whether every class classified as **Functor** has at least one of the classes classified as **Variable** or **Constant** as its part. Again, `forallDebugOne` reports which particular **Functor** class (i.e., `SOULList`) fails to satisfy this architectural constraint.

For reasons of efficiency, the primitive second-order Prolog predicate `forall` fails immediately after the first generated value fails to satisfy the provided logic expression. The same holds for `forallDebugOne`. However, when trying to identify what caused the conformance conflict, it may be useful to find out all values that have failed (rather than only the first one). In that case, we should use the `forallDebugAll` predicate instead (see Table 5.10 on page 78 and see page 111).

This predicate will check all values, even if the first one already failed. Although this predicate generates more useful information, it will be less efficient than `forall` and `forallDebugOne`. (Only in the case of success, they will be equally efficient: everything needs to be checked.) So there is a bit of a trade-off between efficiency on the one hand and obtaining more debugging information on the other.

It should be left as an option (e.g., a user setting) which of the above variants of the conformance checking algorithm to use. If we only want to know whether conformance is valid or not, without further details, the original algorithm should be preferred. If we want to see, in case of non-conformance, precisely what went wrong, we should use the approach with special quantifier predicates. Depending on whether we want to see all problems at once, or just the first one, we can again choose which variant we prefer. (Technically, we implemented this by just overriding the primitive `forall` predicate by whatever version was preferred.)

7.5 Conclusion

In the light of our thesis statement that *"automated support for checking conformance of the implementation of a software system to its architectural views, can be achieved in a very expressive way by adopting a logic meta-programming approach"*, we draw some conclusions from the case study presented in this chapter. In addition to clarifying the architecture language and conformance checking algorithm explained in the previous chapters, the main goal of this case study was to prove the feasibility (see Subsection 7.5.1) and expressiveness (see Subsection 7.5.3) of our approach. In particular, we wanted to show that LMP not only provides an elegant way of implementing the architectural model and conformance checking algorithm (see Subsection 7.5.2), but more importantly, that it provides an expressive medium in which an architect can define the architectural mapping of architectural concepts and relations to implementation artifacts and their dependencies.

7.5.1 Feasibility

The performed case study shows that our consistent use of a LMP language throughout all abstraction layers — from the different layers of the DFW over the architectural abstraction and architectural instantiation to the conceptual architecture — provides a viable formalism to reason about architectural knowledge at a sufficiently high level of abstraction while still allowing conformance checking of source code.

Virtual classifications proved their worth as suitable abstractions of architectural concepts. They hide the details of the lower-level implementation artifacts on which they are mapped, yet allowing us to reason about their relationships with other architectural concepts independently of the artifacts they actually contain. Virtual dependencies provide a powerful way of defining highly abstract relationships among architectural concepts, by building them up from lower-level relationships that are again constructed from even lower level ones. As such, simple low-level relationships can be successfully combined into complex high-level relationships. Based on these mappings of architectural concepts to lower-level artifacts, and of architectural relations to lower-level relationships, it was easy to implement the conformance checking rules by implementing conformance checking at a high level in terms of conformance checking rules at lower levels.

We succeeded in checking conformance of the actual implementation of the SOUL system to the architectural views introduced in Chapter 4. Whereas the 'application architecture' view mapped more or less directly to the implementation structure of the SOUL system, the 'rule-based interpreter' and 'user interaction' view provided examples of cross-cutting architectural mappings. Although the case study illustrated the feasibility of the approach, there are still many difficulties to be resolved. Chapter 8 discusses some shortcomings and explains what is needed for the approach to be applicable in an industrial context.

7.5.2 Logic programming as implementation medium

Our main motivation for choosing a logic language to implement the conformance checking algorithm and architecture language, came from the observation that the proposed algorithm and architecture language have a strong 'logic flavor'. Let us illustrate this by means of some examples:

- The most striking example is probably the conformance checking algorithm itself, which revolves around the construction of a logic expression that can be evaluated to check for architectural conformance.
- Other examples are the obvious mappings of different kinds of architectural abstractions to concepts in a logic programming language: virtual dependencies correspond to logic predicates, and quantifiers to second-order logic predicates such as `forall` and `exists`.
- In general, because of its declarative nature, a logic programming language seems to be a good choice to represent and reason about architectural knowledge. The main advantage of a

declarative approach over a procedural one is that it is better suited to declare knowledge and provide conceptual definitions. Virtual classifications are a good example of this. Although virtual classifications could be described in either a declarative or a procedural language, describing them in a logic language leads to concise and intuitive definitions.

But the logic language is more than a suitable implementation medium for implementing the architectural formalism. By offering an architect the full power of a logic language, with facilities to access the implementation language and a predefined set of logic predicates at different levels of abstraction, a maximum of expressiveness is achieved. We will elaborate on this in the next subsection.

7.5.3 Expressiveness

It should be clear from the examples in this chapter that our architectural conformance checking formalism is indeed a very expressive one. By providing the full power of LMP to define the architectural mapping, we could easily satisfy all expressiveness requirements put forward in Subsection 3.2.2.

Logic meta programming

To define virtual classifications and virtual dependencies an architect can use the full expressive power of the logic programming language, including logic unification, backtracking, recursion, negation, multi-way querying, and so on. This, in combination with the predefined mapping predicates provided by the declarative framework, allows an architect to declare very complex architectural mappings in a reasonably intuitive and concise way. To reason about implementation artifacts, the LMP language provides access to their full parse-tree representations. These parse trees can be analyzed either by traversing them, by lexically analyzing their string representation, or by using a combination of both techniques.

In comparison to an explicitly enumerated classification, a virtual classification is more intentional, as it provides a concise and intuitive description of which artifacts are intended to belong to it. These descriptions can be based on simple naming or coding conventions; or they can be based on some more complex semantic inferencing by using virtual dependencies that describe complex relationships among implementation artifacts; or they can even be defined in terms of other existing or auxiliary virtual classifications.

Virtual dependencies can represent simple implementation dependencies and coding patterns, as well as more complex interaction protocols and design patterns. Our formalism provides a whole range of predefined virtual dependencies. These can either be used directly to represent architectural relations, or they can be used as building blocks in terms of which to define more abstract and more complex virtual dependencies.

Similarly, a whole range of predefined port filters exists, but nothing prohibits us from defining our own port filters. Of course, these definitions can make use of the predefined port filters and of the full expressive power of the logic programming language.

Requirements

Thanks to the power of LMP, our formalism satisfies all requirements enumerated in the criterion of expressiveness in Subsection 3.2.2:

1. *The formalism should pose no a priori restriction on the kinds of implementation artifacts and architectural entities and relationships that can be considered.*

   A virtual classification can describe any set of implementation artifacts. We can even define heterogeneous classifications that contain a mixture of different kinds of implementation artifacts (e.g., classes, methods, variables). A virtual dependency can codify any relationship, ranging from simple implementation dependencies, to relationships describing complex

collaboration or interaction patterns. Our formalism is only restricted in that we consider object-oriented implementations only, and that we only reason about the static structure.

2. *The formalism should allow for composite architectural concepts and relations.*

   We gave many examples of how complex virtual dependencies were constructed from more primitive virtual dependencies. We also showed how virtual classifications could be defined in terms of more primitive virtual classifications and virtual dependencies. In addition to being able to define virtual classifications and virtual dependencies in terms of other virtual classifications and virtual dependencies, the formalism can easily be extended to allow for composite architectural concepts and relations (see Subsection 6.4.5). Composite architectural concepts and relations are not mapped directly to virtual classifications or virtual dependencies, but are defined in terms of sub-architectures containing other high-level concepts and relations.

3. *The formalism should allow for complex architectural relations.*

   We gave many examples of architectural relations that can deal with transitive closures, interaction and collaboration protocols, naming and coding conventions, design patterns, and so on. To manage the complexity, complex architectural relations are defined in terms of high-level virtual dependencies that are in turn defined in terms of more primitive ones, and so on, until an implementation dependency or a predefined virtual dependency is reached.

4. *The formalism should allow for cross-cutting mappings of architectural concepts to implementation artifacts.*

   Many architectural concepts in the 'rule-based interpreter' and 'user interaction' view have a cross-cutting mapping to the implementation, in the sense that the virtual classification corresponding to such a concept generates a set of implementation artifacts that are distributed throughout the entire implementation structure.

5. *The formalism should support the definition of multiple, potentially overlapping, architectural views on the same software system.*

   Our case study defines three architectural views on the SOUL software system, and illustrates how architectural conformance can be checked to each of them. Some of these architectural views are overlapping, in the sense that architectural concepts in different views are mapped to the same virtual classification. Furthermore, many virtual classifications themselves are overlapping in the sense that one implementation artifact can belong to multiple virtual classifications.

7.5.4 Other criteria

In Subsection 3.2.2, we listed a number of criteria that a formalism for architectural conformance checking should satisfy. We already discussed the criterion of expressiveness in the previous subsection. Now we take a look at the criteria of simplicity, efficiency, extensibility and generality.

Simplicity

The proposed formalism is rather simple. The architectural views are defined in a simple ADL, and the mapping of the entities in this ADL to the implementation is also simple: architectural concepts are mapped to sets of implementation artifacts (i.e., virtual classifications), ports to filters over these sets, and architectural relations are mapped to relationships among the artifacts in these sets (i.e., virtual dependencies). Because of its simplicity, the formalism was easy to implement in Prolog. As we saw in section 6.3, the conformance checking algorithm was implemented essentially in a 12-line Prolog rule.

Virtual classifications, which describe how their elements are computed, are more abstract and contain more information than ordinary classifications, that give an explicit enumeration of their

elements. In spite of their more intentional character, our virtual classifications, described in a logic medium, were typically very concise, intuitive and readable.

However, as virtual classifications and virtual dependencies can make full use of the power of LMP, the architect needs to know the LMP language well. To simplify the declaration of such virtual classifications and virtual dependencies, he or she can use a whole range of predefined logic predicates provided by the DFW. In addition, in Subsection 8.3.4 we will discuss a number of tools that could help the architect with the complex task of declaring the architectural mapping.

Efficiency

As can be seen from the timings in the previous sections, our conformance checking algorithm is not so efficient. This is mainly due to the fact that a lot of implementation artifacts are involved, and that the virtual dependencies often require some heavy computation. Also, our use of a logic language may not have been the best choice from the viewpoint of efficiency. (Remember, however, that efficiency was not our major concern. We were mainly interested in an approach that was as expressive as possible. For this purpose, the use of a logic language was a good choice.)

To make the approach more efficient, in addition to the optimizations discussed in Subsection 6.3.3, in Chapter 8 we propose some more optimizations (Section 8.2), as well as a more incremental version of the conformance checking algorithm (Section 8.1).

Extensibility and generality

We deliberately kept the design of our formalism as simple as possible. Some interesting extensions, such as support for architectural styles, deviations, correspondences and sub-architectures, were already discussed in Section 6.4. Some more generalizations will be discussed in Chapter 8:

- In Section 8.1 we explain how the current conformance checking algorithm could be extended into a more incremental version. When changes are made to either the implementation or the architecture, this incremental version re-checks conformance only for those places that were affected by these changes.
- In Section 8.4 we motivate that the current formalism is not specifically tuned towards checking conformance of Smalltalk implementations to software architectures. It could equally well be used to check architectural conformance of implementations in other programming languages (object-oriented as well as others), or even to check architectural conformance of design models.

Chapter 8

Towards an Industrial-Strength Tool

The architectural formalism and conformance checking algorithm we proposed, and the prototype tool we implemented, are still in an experimental stage. To be applicable in an industrial context, the conformance checking formalism and tool should be improved and extended in many ways. A first important extension is to allow for incremental conformance checking. Then we discuss some optimizations that can make our algorithm more efficient. Next, we explain how our conformance checking tool could fit in an industrial-strength tool to support architecture-driven development. We conclude with some future generalizations of the formalism and algorithm.

8.1 Incremental conformance checking

In Subsection 2.1.6, we explained why evolution of both the implementation and the architecture of a software system are essential and unavoidable. After evolution of either the implementation, the architecture or the architectural mapping, conformance checking techniques are needed to verify whether the implementation still conforms to the architecture. If we assume that the implementation was conform to the architecture before the evolution, it is overkill to re-check conformance entirely. Especially for large software systems and complex architectural mappings, the overhead of re-checking conformance entirely may be inhibiting. Therefore, in the context of evolution, an incremental conformance checking approach is more appropriate. With an incremental approach, instead of re-checking conformance for the entire implementation and architecture, we only need to analyze the parts that have evolved.

In this section we sketch how the architectural formalism of Chapter 5 could accommodate such a more incremental approach. The approach is based on a categorization of possible evolutions that can occur. For each possible evolution, we analyze its potential impact on architectural conformance, and draw our conclusions regarding incremental conformance checking.

8.1.1 Kinds of evolution

We assume to start from an initial situation (as depicted on the left in Figure 8.1) where we have an implementation of some software system, a description of its conceptual architecture, and an architectural mapping between them (also see Figure 5.1). Any of these can evolve. We also assume that, in the initial situation, the implementation conforms to the architecture. We are interested in the impact of an evolution on this conformance.

In this subsection, we present a taxonomy of the kinds of evolution in our formalism (see Figure 8.1). For each of these kinds of evolution, we enumerate the different kinds of changes that can

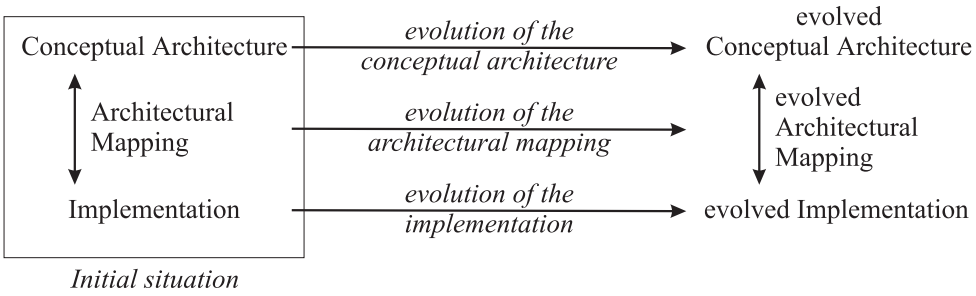

Figure 8.1: Possible evolutions of implementation and architecture.

occur. We will describe these possible changes as 'evolution operators' and illustrate them with examples from the 'user interaction' architectural view.

Evolving the conceptual architecture

With respect to incremental conformance checking, evolving the conceptual architecture is probably the simplest kind of evolution. After making changes to the conceptual architecture, we need to verify whether the implementation still conforms to it. If we know which parts of the architecture evolved and how, we can easily assess how this affects architectural conformance. Because architectural conformance checking compares architectural descriptions with the implementation, most changes to the architecture typically have only a local impact on architectural conformance: conformance only needs to be re-checked for those architectural descriptions that have been changed.

Architectures can evolve on three levels: on concepts or relations, on architectural views as a whole or on the level of the entire conceptual architecture.

- For architectural concepts, we define two evolution operators. *Concept extension* extends the 'interface' of an architectural concept by adding a new port to it. *Concept cancellation* removes a port from some concept. The evolution operators for relations are similar. *Relation extension* extends the 'interface' of a relation by adding a new role to it. *Relation cancellation* removes a role from that relation.

  For example, originally, the User Application concept in the 'user interaction' view only had one port corresponding to classes representing user applications in the SOUL system. Later we added an extra port (i.e., concept extension) corresponding to the methods of these classes, to represent the requests that can be handled by SOUL user applications.

- On architectural views, we can define the following useful evolution operators. *View extension* extends an architectural view by adding extra concepts or relations to it. The added elements are not connected to any other element. *View cancellation* removes (unconnected) concepts or relations from an architectural view. *View refinement* refines an architectural view by adding extra links between the concepts and relations in it. *View coarsening* removes existing links between concepts and relations in the architectural view.

  For example, in the 'user interaction' view, originally we did not make a distinction between user applications and auxiliary applications. The concept Auxiliary Application was only introduced later (i.e., view extension) when we realized that there are some applications which are created and activated by others in response to certain user requests.

- Finally, on conceptual architectures, which consist of multiple views, we also have two evolution operators. *Architecture extension* extends a conceptual architecture with a new architectural view. *Architecture cancellation* removes an architectural view from the conceptual architecture.

> Again, the 'user interaction' view is a good example. Initially, we defined only two architectural views on SOUL, namely the 'rule-based interpreter' and 'application architecture' view. The 'user interaction' view was only added later (i.e., architecture extension) because we needed a view that focused on the user-interaction aspects of SOUL.

In Subsection 8.1.2, we will analyze the potential impact of these architectural evolution operators on conformance of the implementation to the architecture.

The choice and terminology of the evolution operators are strongly inspired by the research on *reuse contracts* [40, 41, 53, 78], where we also use the evolution operators *extension*, *cancellation*, *refinement* and *coarsening*. The technique of reuse contracts supports the detection of evolution conflicts when two independent changes are made to the same software artifact. This is done by comparing the evolution operators that describe the two evolutions, for potential unexpected interactions. Furthermore, the above taxonomy of evolution operators on architectures is very similar to the one proposed by N. Romero [71], although she used a slightly different terminology. N. Romero applied the technique of reuse contracts to manage the evolution of software architectures.

It should be noted that the evolution operators enumerated above are not necessarily independent. For example, a view coarsening may require a relation cancellation to retain a consistent architecture. Indeed, an architecture which has a relation with a role that is not linked to any port is not well formed. Also, an evolution of the architecture will often require an evolution of the architectural mapping as well. The next subsection enumerates the possible evolution operators for the architectural mapping.

Evolving the architectural mapping

Recall that the architectural mapping is split into an architectural instantiation and an architectural abstraction.

Architectural instantiation Evolving the architectural instantiation typically has a local impact. It affects only the architectural entities for which the instantiation is changed. In essence, checking conformance after changing the architectural instantiation will require re-checking only these entities (see Subsection 8.1.2).

The architectural instantiation maps concepts, ports, relations, roles and links to elements of the architectural abstraction. For the architectural instantiation, the only kind of evolution that is allowed is to replace the instantiation of some architectural entity by another instantiation, in other words to 're-define' the entity. We distinguish the following kinds of re-definitions: *concept re-definition*, *port re-definition*, *relation re-definition* and *link re-definition*. For example, a concept re-definition replaces the virtual classification associated with a concept by another one. We also have a *roles re-definition*: it takes a *set* of roles as input, and permutes the associated argument numbers.

To illustrate this, let us return to the example of the User Application concept in the 'user interaction' view. As mentioned above, the Auxiliary Application concept was only introduced later. At that moment, however, it was necessary to redefine the instantiation for the User Application concept, so that it only referred to classes that did not represent auxiliary applications. This involved a *concept re-definition* to instantiate the User Application concept with this new definition.

Architectural abstraction Evolving the architectural abstraction has a larger impact, since the abstractions it defines can be used in the instantiation of multiple architectural entities, or even in the definition of other abstractions. Nevertheless, to some extent, we can still re-check conformance incrementally, if we know exactly *how* the architectural abstraction changed. Therefore, it is important to categorize the possible ways in which the architectural abstraction can evolve.

The architectural abstraction defines virtual classifications, filters, virtual dependencies and quantifiers in terms of logic predicates. Therefore, at this level, we consider the evolution operators *extension*, *cancellation*, *strengthening*, *weakening* and *re-definition*, which correspond to the typical changes that can be made to logic predicates.

It is clear that we need evolution operators for introducing new (i.e., extension), or removing existing (i.e., cancellation) virtual classifications, filters, etc. When removing things we assume they are not referred to anymore.

More interesting evolution operators are re-definition, strengthening and weakening. Re-definition changes the definition of an existing predicate. Strengthening a logic predicate 'restricts' the predicate so that it has less solutions than before. Weakening a logic predicate 'relaxes' that predicate so that it has more solutions than before. Now, let us take a closer look at these evolution operators:

- *Virtual classification strengthening* 'removes' artifacts from a virtual classification so that the set of artifacts it computes is a subset of the original one.

  For example, reconsider the example where we re-defined the User Application concept, by first defining a new virtual classification (i.e., virtual classification extension) and then replacing the old virtual classification by this new one (i.e., concept re-definition). We could equally well have achieved the same effect by using a virtual classification strengthening which does an 'in place' redefinition of the virtual classification of User Application so that it does not produce classes that represent auxiliary applications anymore. The difference with a concept re-definition is that a virtual classification strengthening does not define a new virtual classification, but re-defines an existing one. (Note that the impact of performing a virtual classification strengthening may be higher, for example, when the same virtual classification is used to instantiate more than one concept.)

- *Virtual classification weakening* 'adds' artifacts to a virtual classification, so that the classification it computes is a superset of the classification computed by the original one.

  For example, the 'queryInterpreter' classification (in terms of which the Query Interpreter concept is defined), originally included only those methods that belong to a method protocol named 'interpreting'. Later we added all methods that belonged to the 'interpretation' and 'unification' protocols as well.

- *Virtual classification re-definition* is a mixture of virtual classification strengthening and weakening, where some artifacts are added to the classification, and some others are removed.

- *Filter strengthening* makes a filter more restrictive than before (e.g., by adding an extra condition to it). All artifacts that are accepted by the new filter, will also be accepted by the original one, but not vice versa.

  For example, at the moment, the 'Request' port on User Application returns all methods of classes that represent user applications. But we are not really interested in all those methods, only in those that represent useful requests for user applications. Changing this would require a filter strengthening.

- *Filter weakening* makes a filter less constraining than before (e.g., by removing one of its filtering conditions). It accepts all artifacts that were accepted before, and maybe some more.

- *Filter re-definition* is a mixture of filter strengthening and filter weakening. The filter is changed such that some artifacts that were accepted before are not accepted anymore, but some others that were not accepted before are accepted now.

- *Virtual dependency strengthening* strengthens the logic predicate representing a virtual dependency. Everything that holds for the strengthened predicate still holds for the original version but not vice versa. Strengthening shrinks the solution set of the predicate.

For example, the `asks_C_M` virtual dependency originally checked only whether the 'Interrogator' class invoked the 'Interrogated' method. Later, we strengthened this virtual dependency to verify whether the result returned by the 'Interrogated' method was actually used by the 'Interrogator'.

- *Virtual dependency weakening* weakens the predicate representing a virtual dependency. A predicate is weaker than another one, if every solution to the original predicate is also a valid solution to the modified one, but not vice versa. In other words, the solution set of the predicate is enlarged.
- *Virtual dependency re-definition* is a mixture of virtual dependency strengthening and weakening, such that some tuples that were dependent before, are not anymore, whereas some others that were not, are dependent now.

This concludes our taxonomy of evolution operators for the architectural mapping. An analysis of the potential impact of these operations on architectural conformance follows in Subsection 8.1.2.

Evolving the implementation

After evolving the implementation, we need to check whether conformance to the architecture is still valid for the evolved implementation. At first sight, it may seem that by categorizing the different kinds of implementation evolution operators and by comparing these evolution operators with the architectural mapping, we can get an idea of the potential impact of this implementation evolution on architectural conformance. However, as the architectural mapping makes use of a full LMP language, assessing this impact is far from trivial. Furthermore, because there is no simple one-to-one mapping of implementation artifacts to architectural entities, re-checking conformance can become problematic. One implementation artifact can correspond to many architectural concepts and can be involved in many architectural relationships. Similarly, one implementation dependency may be used to define many architectural relations or architectural concepts. As a consequence, it may be necessary to re-check conformance for large fragments of the software architecture. Nevertheless, in the next subsection, we will propose a tentative solution for incrementally checking architectural conformance when the implementation evolves.

8.1.2 Analyzing the impact on architectural conformance

Now that we have discussed the different kinds of evolution, for each evolution operator we analyze its impact on architectural conformance and draw conclusions on how conformance can be checked incrementally. We only discuss this informally. At the time of writing, an incremental conformance checking algorithm has not yet been implemented.

Impact of evolving the architecture

The situation for evolution of the architecture is rather simple. Incremental conformance checking boils down to performing local re-checks of some architectural descriptions. For some kinds of changes, we do not even have to re-check anything. Below, we assess the possible impact of architectural evolution on architectural conformance, by analyzing the different kinds of architectural entities that can be modified: concepts, relations, views or the entire conceptual architecture. We further split up our analysis based on the possible evolution operators for those entities.

- *Concept extension and cancellation* have *no impact* on architectural conformance, as adding ports to or removing ports from an architectural concept is only allowed for ports that are not linked to anything. When a port is not linked to any architectural relation, this means that it does not participate in any architectural constraint, and thus does not pose any particular constraint on the implementation. E.g., adding a 'Request' port to the User Application concept had no impact on architectural conformance, as long as it was not linked to any architectural relation.

- *Relation extension or cancellation* adds a new role to or removes an old role from an architectural relation. As a consequence, the virtual dependency associated with the relation will need to be replaced by a new one that takes the extra or removed argument into account. Although the added or removed role is not allowed to be connected to anything else, the other roles of the relation may be, so the new predicate needs to be checked again anyway. (After having linked the added role, in case of an extension.)

  As an example, consider the Creates With architectural relation in the 'user interaction' view of Figure 7.1. Initially, this relation had only two roles: the 'Creator' role representing a user application that needs to render some output, and a 'Created' role representing the output viewer on which the output would be rendered. Later we added a third role 'Argument' representing the type of output that needs to be rendered. Obviously, after making this change, we also needed to replace the virtual dependency associated with Creates With by one that took three arguments instead of two. To verify whether the implementation was still conform to this changed architectural view, we only needed to re-check the changed Creates With architectural relation.

- *View extension and cancellation* are only allowed if the architectural concept or relation to be removed or added is not linked to any other element. Therefore, there is *no impact* on architectural conformance, as the element does not participate in any architectural relation. E.g., adding a new architectural concept Auxiliary Application obviously does not affect architectural conformance, as long as it is not linked to anything.

- *View refinement* adds links between architectural elements and requires re-checking the architectural relation to which a link has been added. E.g., after introducing the Auxiliary Application concept, we linked it to the (existing) Activates relation with the Input Window concept. To verify conformance, we needed to re-check conformance of the implementation to the Activates relation (but only for the Auxiliary Application concept).

- *View coarsening* removes a link from a role of some architectural relation. To analyze the impact of this change on architectural conformance, we need to distinguish two cases. Either the role had only one link attached to it, or it had multiple links attached. (In a well-formed architecture, every relation role is supposed to have at least one link attached to it.) In the former case, removing the link is not allowed (unless in the exceptional case that we simultaneously remove the links to all other roles of the same architectural relation) because it would yield a malformed architecture. In the latter case, after the link removal, the architectural relation needs to be re-checked: as multiple outgoing links on the same role represent a disjunction, the semantics will change by removing one link.

  For example, suppose that we would want to remove the link from the 'Action' role of the Activates relation to the 'Request' port of the Auxiliary Application concept (in Figure 7.1). This is allowed, as the 'Action' role is still linked to the 'Request' port of the User Application concept. However, after this removal the Activates relation needs to be re-checked as it may not be valid anymore. (The fact that an Input Window activates at least one Auxiliary Application *or* at least one User Application, does not imply that it will activate at least one User Application.)

- *Architecture extension* obviously requires checking conformance for the entire added architectural view, but nothing more. E.g., when we introduced the 'user interaction' view, we had to check conformance of the implementation to this view, but we did not have to re-check the other views.

- *Architecture cancellation* has *no impact* on architectural conformance. It merely deletes a whole set of architectural constraints, described by the removed architectural view.

Impact of evolving the architectural instantiation

As evolving the architectural instantiation has a rather local impact, it requires only some local re-checking of architectural conformance. *Relation, link and roles re-definition* require re-checking architectural conformance for the relevant architectural relation only. *Concept and port re-definition* require re-checking conformance for all architectural relations in which the affected concept participates.

- *Concept re-definition* changes the instantiation of some architectural concept and may affect all architectural relations in which that concept participates.
- *Port re-definition* changes the instantiation of a concept's port and may affect all architectural relations that are linked to that port.
- *Relation re-definition* only affects the particular architectural relation for which the architectural instantiation is changed.
- *Link re-definition* only affects the architectural relation to which this link is connected.
- *Role re-definition* only affects the architectural relation to which this role belongs.

Before, we gave the example of re-defining the User Application concept. This would require re-checking all relations to which it is connected. Unfortunately, this particular concept is connected to all relations in the 'user interaction' view, so we need to re-check every relation (in this view). This is not the case for all concepts, however. If we re-define the Input Window concept, for example, then we only need to re-check the Activates relation.

Impact of evolving the architectural abstraction

When defining new elements in the architectural abstraction (i.e., extension), or removing existing ones that are not referred to anymore (i.e., cancellation), there is obviously *no impact* on architectural conformance. The other evolution operators, however, which change an element of the architectural abstraction, may have a large impact. The changed element might be used in many instantiations, as well as in the definition of many other elements. Below, we explain for these evolution operators how architectural conformance can still be checked incrementally.

- *Virtual classification strengthening, weakening and re-definition* affect every architectural concept that is instantiated with this virtual classification, and therefore also every architectural relation in which such a concept participates. All these architectural relations need to be re-checked. Furthermore, all virtual classifications that are defined in terms of the changed one, are also affected. To analyze the impact of these affected virtual classifications, a similar analysis can be made.

  Although we only need to re-check those architectural relations in which the affected architectural concepts participate, this solution still seems rather inefficient. Thanks to our good choice of architectural abstractions and the use of LMP, however, re-checking an architectural relation can often be done much more efficiently than may seem at first sight. More specifically, there is no need to re-check the virtual dependency associated with the architectural relation for *all* artifacts of the virtual classification associated with the affected concept. It suffices to re-check the architectural relation for some artifacts only, taking the semantics of the quantifiers into account.

  For example, suppose that some architectural concept is instantiated with a virtual classification which was *weakened*. This means that the virtual classification now computes some more artifacts than before. Assume further that this concept participates in some architectural relation, and that it is linked to that relation with a $\forall$ quantifier. To verify conformance for this relation we should check whether the virtual dependency to which it is mapped is

valid for *all* artifacts generated by the virtual classification. However, under the assumption that conformance was already valid before the weakening, we know that the virtual dependency already holds for all original artifacts in the virtual classification. Therefore, we need to check the virtual dependency *only* for the new artifacts in the classification. Obviously, this is much more efficient than re-checking it for all classified artifacts. If the quantifier attached to the link is an $\exists$ quantifier, the situation is even better. The assumption that conformance was already valid before the weakening implies that there *already existed* some artifact that satisfied the virtual dependency. As adding an artifact to the classification does not change this situation, *nothing* needs to be checked. (Our earlier example of weakening the 'queryInterpreter' classification to add all methods belonging to the 'interpretation' and 'unification' method protocols, illustrates this. Nothing should be re-checked, as the Query Interpreter concept is only linked to a relation by means of an $\exists$ quantifier.)

As a second example, suppose that we are dealing with a *virtual classification strengthening* instead of a weakening. This means that the virtual classification is more restricted and will produce less artifacts than before. Assume that some concept, instantiated with this classification, participates in some architectural relation, using a $\forall$ quantifier. If conformance was already valid, it will remain valid: if the virtual dependency, associated with this relation, already succeeded for all artifacts, it will still succeed for all remaining artifacts after the strengthening. Therefore, *nothing* needs to be re-checked. However, when the quantifier is an $\exists$, the situation is a bit more subtle. If the 'removed' artifacts were not relevant for the (existence of the) virtual dependency, there is no impact. But if we remove the artifact for which the virtual dependency existed, conformance will fail, unless there still exists another artifact for which the virtual dependency holds. In that case, we should look for such an artifact in the changed virtual classification. Note that, to know whether or not the 'removed' artifacts affect the relation, we need a 'memoized' version of $\exists$ which remembers the value(s) for which it succeeded.

- *Filter strengthening, weakening and re-definition* affect every architectural relation linked to a port that is instantiated with this filter. All these relations need to be re-checked. Other filters that are defined in terms of this changed one, are also affected. For these affected filters, we can make a similar reasoning to assess their impact.

  Again, re-checking an architectural relation can be done more efficiently than may seem at first sight. For example, a *filter strengthening* has the effect of shrinking the set of artifacts generated by a port. By taking the semantics of the quantifiers into account, we can re-compute the architectural relation more efficiently. The situation is similar to that of a *virtual classification strengthening*: in both cases the set of artifacts shrinks. Analogously, a *filter weakening* enlarges the set of generated artifacts and is similar to a *virtual classification weakening*.

- *Virtual dependency strengthening, weakening and re-definition* affect every architectural relation that is instantiated with this virtual dependency. Furthermore, it affects all virtual classifications that use this virtual dependency in their definition (which will in turn affect some concepts, relations and other virtual classifications, as explained above). It also affects all virtual dependencies that use this virtual dependency in their definition.

  It is a non-trivial problem to find all affected virtual classifications and virtual dependencies. One possible solution is to use some kind of dependency analysis in the logic language to find all predicates that depend on a certain predicate.

  For the affected virtual classifications, checking incremental conformance can be done similar to what was described above. We first re-compute each affected classification and compare it with its original version, to get an idea of which artifacts have been added, and which have been removed from this classification. Then we find the architectural concepts that use this classification, and re-check all architectural relations in which these participate. Based

on our knowledge of the added and removed artifacts, this can be done rather efficiently, by taking into account the semantics of the quantifiers.

For the affected virtual dependencies, we need to re-check all architectural relations that are instantiated with such a virtual dependency. Again, we investigate whether this re-check can be made more efficient, for example, by re-checking only those tuples for which the relation has changed. A first solution would be to compare the solution set of the changed virtual dependency with the solution set of its original version. Based on the tuples that have been added to or removed from this set, and using the semantics of the quantifiers, we can re-check only what is necessary. However, this is not really an optimization as computing the solution set entirely is precisely what we wanted to avoid. A second solution is to use second-order predicates that are a bit more intelligent. For example, if we have a 'memoized' $\exists$ quantifier which 'remembers' the artifacts for which the virtual dependency existed, we can first try these artifacts. If we are lucky, the modified virtual dependency still holds for at least one of the same artifacts. If not, we do have to re-check the virtual dependency for all remaining artifacts. In case of a $\forall$ quantifier, the situation is less advantageous, as we have to re-check the virtual dependency for all artifacts.

Impact of evolving the implementation

As already mentioned earlier, changing the implementation has probably the highest impact on architectural conformance. Mainly because of the cross-cutting nature of the architectural mapping and because the architectural mapping makes use of a full-fledged LMP language, every small implementation change can affect this mapping in many ways. For example, when adding a new method to the implementation, this change may affect every architectural abstraction that (directly or indirectly) reasons about this method or parts of it.

To be able to incrementally check architectural conformance when the implementation evolves, we propose to use an *incremental constraint solving* approach. We only mention this approach very briefly here, and refer to R. Wuyts' Ph.D. dissertation [87] for more technical details. In the context of his Ph.D. research, in order to experiment with techniques for synchronizing design and implementation, R. Wuyts implemented an incremental symbolic constraint solver in Smalltalk.

In this approach, instead of considering the architecture as a logic query that reasons about the implementation, it is viewed as a set of constraints on the implementation, that is checked when the implementation changes. An incremental symbolic constraint solving algorithm that combines techniques from constraint logic programming and numeric incremental constraint solvers is used to re-check only those constraints that are affected when a certain change is made to the implementation. Incremental constraint solving means that the results of a previous run are not just discarded, but are used when domains of variables change. These changes are then propagated in order to update all the domains of the variables affected by the initial change. To make sure that changes in the implementation affect the constraints expressed in the constraint network, every time the implementation is changed, the incremental constraint network is triggered to propagate the necessary changes.

As a concrete example, suppose that we add a new method to the implementation. This implementation change may affect every logic predicate of the representational mapping that reasons about this method are parts of it. These predicates are used to define higher-level predicates, which are in turn used to define virtual classifications and virtual dependencies, which are eventually used to define the architectural constraints. To assess the impact of the implementation change on these architectural constraints, we simply propagate the change through the constraint network, until everything that is affected by the change has been re-checked.

It remains to be investigated whether the approach sketched above is feasible for our purposes, though. As the constraint network needs to remember all previous results, the amount of memory required to store all these results may be too high. And even if it would be possible to apply the approach on our current case study, the question remains if it scales to larger software systems.

8.1.3 An example of architectural evolution

To illustrate part of the incremental conformance checking algorithm, we work out an example of an interesting architectural evolution of the 'user interaction' view. Until now, the Asks_1 relation was modeled as a binary relation between the User Application and Query Interpreter concepts. In footnote 4 on page 46 we mentioned that representing this relation as a ternary relation linking User Application, Query Interpreter and Query Result, would better reflect our intuition that a user application asks a query interpreter to compute some query result. Making this change results in the evolved architectural view depicted in Figure 8.2. This figure differs from Figure 5.2 in that Asks_1 has an extra role 'Result' which is linked to the 'Type' port on Query Result. The link has an $\exists$ quantifier attached to it.

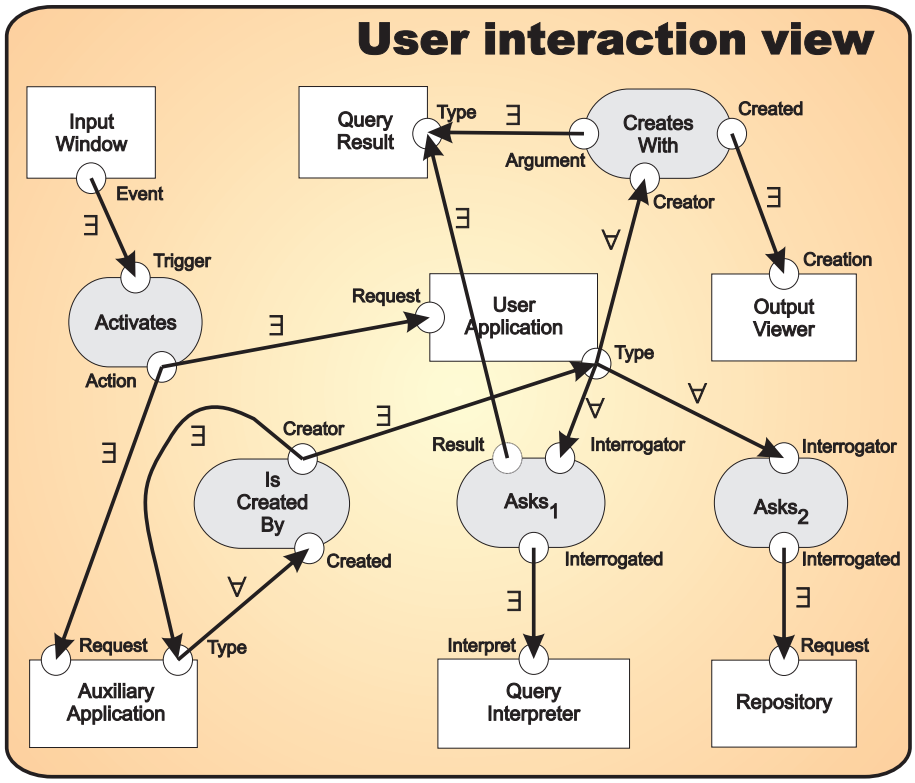

Figure 8.2: An evolved version of the 'user interaction' view.

To analyze the impact of this architectural evolution, we first decompose it in terms of primitive evolution operators. The following evolution operators are needed:

1. a *virtual dependency extension* to introduce a new predicate `asks_C_C_C(Interrogator, Interrogated, Result)` which checks whether some `Interrogator` class invokes some `Interrogated` class which returns a result of type `Result` to the `Interrogator` class.

2. a *relation extension* to add a 'Result' role to the Asks_1 relation.

3. a *relation re-definition* to replace the old virtual dependency which Asks_1 was instantiated with by the new one `asks_C_C_C`.

4. a *view refinement* to link the Asks_1 relation to the Query Result concept. More specifically, the 'Result' role of Asks_1 is linked to the 'Type' port of Query Result, with an associated quantifier $\exists$.

Using the analysis of Subsection 8.1.2, we can know assess the impact of these evolution operators on architectural conformance. The *virtual dependency extension* has no impact on architectural conformance, as it introduces a new virtual dependency that is not yet used by anything. The *relation extension* adds a new 'Result' role to Asks_1, and requires the original virtual dependency associated with Asks_1 to be replaced by the new one. This is done by the *relation re-definition*. Because Asks_1 is now instantiated with a new virtual dependency, it should be re-checked, after having linked its new 'Result' role. This is done by the *view refinement*. Our careful analysis of Subsection 8.1.2 stipulates that this *view refinement*, *relation re-definition* and *relation extension* only require re-checking conformance of the implementation to the evolved Asks_1 architectural relation.

8.1.4 Conclusion

We categorized the different kinds of changes that can be made to the conceptual architecture, architectural mapping and implementation, and analyzed the impact of these changes on architectural conformance. Using this analysis, we sketched a more incremental version of our conformance checking approach that does not re-check conformance entirely, when changes are made, but only re-checks those parts that are affected by the change.

As expected, architectural changes (including changes to the architectural instantiation) lend themselves better to incremental conformance checking than changes to the architectural abstraction and implementation changes. The latter typically have a much higher impact due to the fact that architectural views may cross-cut the implementation. One implementation artifact may address several architectural concepts and one implementation dependency may be used in the definition of multiple architectural relations and concepts.

Our formalism has some nice properties that facilitate incremental conformance checks. To mention just a few, we can take advantage of multi-way reasoning to use a virtual classification for checking instead of generating purposes, and we can make intelligent use of the quantifier predicates to avoid re-checking architectural relations entirely. Again this confirms our good choice of architectural abstractions and of adopting a LMP approach.

Some extra reasoning mechanisms still need to be added to our logic medium, however. More particularly, we need a mechanism for computing all logic predicates that are dependent on a given predicate. We also need an incremental constraint solver for assessing the impact of implementation changes on architectural conformance. Another mechanism that would be useful is an automated synchronization mechanism between the logic meta language and the object-oriented base language. This would allow the incremental conformance checking algorithm to be triggered automatically whenever a change is made to the implementation. In the context of his Ph.D. dissertation, which focuses on techniques for automated synchronization of object-oriented implementations with design information codified in a LMP language, R. Wuyts is currently experimenting with such mechanisms [87]. It is outside the scope of our dissertation to discuss these mechanisms in more detail.

At this point, we did not yet implement, nor experiment with, our incremental conformance checking algorithm. Before doing so, we think it is best to perform some more elaborate case studies in an industrial context. As an alternative to an incremental approach, we may still use the original conformance checking algorithm and run it in background, or overnight on a fast computer. The case studies could demonstrate whether or not this approach is feasible and usable in practice.

One interesting extension to make the incremental approach more practical would be to allow submitting a whole set of changes *simultaneously*. The incremental algorithm sketched above assumes that changes are made one at a time. Allowing multiple changes at the same time has the advantage of allowing the implementation, during the course of these changes, to be temporarily inconsistent with the architecture. This extension to the incremental algorithm requires some careful investigation though, because the temporary inconsistency may cause extra complications.

An even further extension is to provide support for *co-evolution*, i.e., allowing multiple simul-

taneous changes at different levels (for example, to both the implementation and the architecture). This may prove even more difficult, as changes are being made at two levels of abstraction at the same time. At this point, it has not been investigated how the incremental approach should be extended to deal with this.

8.2 Further optimizations

Incremental conformance checking can be seen as an interesting optimization of the conformance checking algorithm explained in Section 6.3 . In this section we discuss some further optimizations and extensions that are valid for both the original and the incremental conformance checking algorithm.

More detailed conflict information

In Section 7.4, we discussed the need for the conformance checking algorithm to generate more detailed information (rather than just a 'yes' or 'no') in the case of failure of a conformance check. This information is necessary to be able to resolve encountered conformance conflicts. As explained in Subsection 7.4.2, the original algorithm could be extended easily to support this, by using 'debugging' versions of the quantifier predicates (such as `forallDebugOne` and `forallDebugAll`) that report failures to the user.

Optimizing time-efficiency by means of caching

One of the most important problems with conformance checking is the large amount of computation time required. This is mainly due to the vast amount of data involved (i.e., all implementation artifacts), the combinatorial explosion of possible relationships and the cross-cutting nature of the architectural mapping. In this subsection we discuss a number of optimizations that improve the time-efficiency at the cost of increased memory usage. They are all based on some form of *caching*.

1. We cache the set of artifacts that belong to some virtual classification, so that the classification does not need to be recomputed every time it is needed (see Subsection 6.3.3).

2. We cache the results of the most frequently-used virtual dependencies (see Subsection 6.3.3).

3. Just like we implemented special debugging versions of the quantifier predicates, we could implement 'memoized' versions of the quantifier predicates that remember their results. This optimization was already discussed in the incremental conformance checking algorithm, to improve the time-efficiency of re-checking architectural relations after the implementation or architectural abstraction had been modified. For example, the original `exists` quantifier predicate merely checks whether at least one generated value satisfies a certain logic expression. The memoized version of `exists` also remembers for which values the expression holds. We explained in Section 8.1 how this extra information is of use to quickly assess the impact of an implementation change on the architectural constraint expressed by the quantifier predicate.

The three examples given above are only an indication of the efficiency gains that can be achieved through caching. It is future work to investigate where and how the time-efficiency of the conformance checking algorithms can be improved even more by using such techniques. As always, there is an important trade-off to be made here. Although caching may improve the time-efficiency, it typically increases the amount of data that needs to be stored. Therefore, only the most computationally-intensive parts should be cached.

Optimized reasoning mechanisms

In all experiments we conducted, we used a Prolog-like LMP language. The disadvantage of such a language is that it is restricted to one single search strategy: depth-first search [44]. Many other strategies (e.g., breadth-first search) are imaginable, however. In addition, Prolog uses a goal-driven reasoning strategy, invoking the rules backwards. Starting from a desired goal, only those rules that lead to this goal are tried out. Every clause in the body of such a rule is a new goal that needs to be resolved. A data-driven reasoning strategy, on the other hand, resolves

the rules in the forward direction, starting from an initial sets of facts, deriving more and more facts in every step, until no more facts can be derived or until some desired solution is achieved. As opposed to Prolog's backward reasoning strategy, many expert systems (e.g., KAN [76]) and constraint solvers use forward reasoning (or a combination of backward and forward reasoning) instead.

Another problem with Prolog, from the viewpoint of efficiency, is its use of unification, a kind of deep-recursive and multi-way pattern matching. Although unification is a powerful mechanism in logic languages, and for reasoning about software architectures in particular (see [47]), for some kinds of usage it is a bit of overkill. For example, instead of using unification, G. Murphy [57] used a conformance-checking approach based on simple GREP-like string-based pattern matching. Although such an approach may be less precise than an approach (such as ours) based on unification over parse trees, it is much more efficient. Therefore, it would be a good idea to include some of Murphy's ideas into our approach. In fact, we already experimented with using string-based pattern matching to optimize some computationally-intensive logic queries. As an example, in Subsection 7.1.7 we discussed the predicate `isUsedBy_E_M`. It first uses string pattern matching to quickly find an expression of interest and then uses a more precise parse-tree search based on unification to refine the found results.

Whereas it is clear that string-based pattern matching can greatly improve efficiency in some cases, it is not obvious how other search and resolution strategies (depth-first, breadth-first, forward chaining, backward chaining, database queries, ...), or other pattern matchers and advanced unification schemes could improve efficiency (or precision). Therefore, we are planning to incorporate and experiment with some of these techniques in our LMP language. Ideally, choosing among all these different techniques should not become a burden for a user of the language. On the long term, we envision having a LMP environment which implements different search and resolution strategies, where the most optimal strategy will be chosen transparently by the system, based on an analysis of the initial query given by the user.

Optimizing memory-efficiency

Although the problem of time-efficiency is an important one, the issue of memory-efficiency should not be neglected either. In fact, the time-efficiency problem is implicitly caused by the vast amount of data that the algorithm needs to process. Since all this data needs to be stored somewhere, it is clear that memory usage is an important concern. Furthermore, a lot of optimizations to increase time-efficiency are based on caching, thus increasing the amount of storage required. For example, although virtual classifications can be stored concisely as predicates that compute a set of values, in the optimized version, all values are stored explicitly in a cache. Similarly, although more high-level dependencies can be computed from more primitive ones, for reasons of time-efficiency, some of them are cached and need to be stored explicitly as well.

Conformance checking compares architectural descriptions with the implementation. All these implementation artifacts and their relationships, as well as all cached values and results, should be stored in some repository that can be accessed by the logic language. Due to memory limitations, the most straightforward approach to store these items as facts in the logic language is not feasible, even for small software systems. Therefore, they have to be stored in an external repository, which has the disadvantage that the access-time is higher. Using a database as an external repository, however, allows even very large systems to be stored and intelligent database queries can be used to retrieve the data efficiently. In this context, it is important to investigate what the most optimal storage scheme is. The data should be stored such that it can be retrieved as fast as possible, without too much redundancy.

8.3 An industrial-strength tool

The prototype implementation we described in Chapter 6 and used in Chapter 7 to conduct our case study, still has many shortcomings. Therefore, in this section we take a closer look at what a good "industrial-strength" tool for checking architectural conformance could look like.

For the sake of the discussion, assume the following scenario. We start out from some software system of which only source code is available. The system has been reasonably well designed and structured, but its architecture was never documented explicitly. Because major modifications to the system are eminent, the management decides to take on the major effort of documenting the system architecture. To assure that this is a one-time effort, a conformance checking tool (such as the one we proposed) will be used to make sure that the source code of the system is and remains conform to this architecture. A senior project member is assigned as software architect. Modifications to the source code that do not conform to the architecture are not allowed. Due to deadline pressure, however, sometimes such modifications cannot be avoided. In such a case, the modifications can only occur with consent of the architect, who explicitly documents these non-conform modifications. When major modifications of the system occur that require an evolution of the architecture, the architect will make the necessary changes, and verify whether the source code still conforms to this evolved architecture. If this is not the case, the developers will be asked to refactor the source code to bring it back in line with the architecture. (At this point, as far as possible, the explicitly documented deviations of the architecture are also brought back in line.) With such an approach, and with an adequate tool to support it, it is assured that the architecture provides an exact picture of the system (modulo some remaining but explicitly documented deviations). At this point, the management obtains its return on investment.[1] Having an up-to-date architecture has many benefits: the re-engineered system has become more maintainable, easier to understand, easier to evolve and reuse, and so on.

Of course, this scenario requires more than progressive managers and an adapted software development process. Equally important is an integrated development environment which enables, facilitates and supports such an architecture-driven development process. In the next subsections, we explain step by step, following the requirements of the above scenario, which supporting tools and techniques are needed in such an environment. But first we analyze the different tasks and activities in the scenario that need to be supported.

8.3.1 Reverse engineering the architecture

Probably the most labor-intensive (and thus most costly) task is *reverse engineering* one or more architectural views from the implementation of the software system. Both the description of the architectural view and its mapping to the implementation should be reverse engineered. The complexity of this reverse engineering task may inhibit the use of a conformance checking approach such as the one we proposed. Before explaining how the reverse-engineering process could be supported by tools, we explain how it was achieved in our case study. From this experience we derive which kind of tool support is useful and desired to support this reverse-engineering process.

Reverse engineering strategies Various strategies can be followed to reverse engineer an architecture from the implementation [4]. Whereas a *top-down* strategy first defines the expected architectural concepts and relations and then tries to find these in the code, a *bottom-up* strategy tries to extract architectural concepts and relations from the code. An *opportunistic* strategy is a mix of a bottom-up and a top-down approach.

[1] On the condition that the cost of extracting and maintaining the architecture and ensuring its conformance with the source code, is lower than the cost of making the changes directly. Note that this cost strongly depends on the availability of good tools to support extraction, conformance checking and maintenance of software architectures. Also note that some hidden costs should also be taken into account. For example, it is typically more difficult to maintain a system if its architecture is ill-designed or unknown.

- For the 'rule-based interpreter' architectural view, we adopted a top-down strategy. This is because the architecture of a rule-based interpreter is well known [4, 31, 74]. As the SOUL system includes a rule-based interpreter, we expected part of its implementation to conform to this architecture. The main developer of SOUL, R. Wuyts, agreed that, from a conceptual point of view, this architecture indeed provided a good description of the rule-based interpretation process. However, he warned that the structure of his object-oriented implementation did not immediately reflect this architecture. Instead, the class decomposition of his software system closely resembled the abstract syntax tree of the logic language. This lead us to defining a cross-cutting mapping from the architectural concepts to the implementation artifacts.

- As the purpose of the 'application architecture' is to give an idea of the global implementation structure, we obviously followed a bottom-up strategy to define this architectural view. The architectural mapping was a straightforward mapping of the architectural concepts to the class structure of the implementation.

- After these two architectural views were defined, we still felt a need for a third additional view which emphasized the 'user interaction' aspect. Here we adopted a more opportunistic approach. Based on our experiences with using the SOUL system, and based on the acquired insights on the structure and workings of the system, we drew an initial sketch of the expected 'user interaction' architectural view. This sketch was refined after discussing it with the main SOUL developer. At that point we tried to map it to the implementation, which triggered some further refinements.

The process of reverse engineering the architectural views for the SOUL system was rather labor intensive, as it was done completely by hand, with little or no tool support. In retrospect, we discuss which manual activities could have been simplified by tools. In Subsection 8.3.4 we will elaborate on a number of tools and techniques that support these activities. Most of these tools are not specifically targeted to support the reverse-engineering process. Amongst others, they may facilitate re-engineering as well.

Declaring architectural views graphically. It would be useful to have some support for declaring architectural views in some graphical notation, which is automatically translated to the corresponding declarations in the LMP language.

Understanding and browsing the implementation. When following a bottom-up strategy to recover some architectural view, or when defining the architectural mapping, a lot of insight in the implementation structure is needed. Therefore, we need sophisticated tools for browsing and navigating through the code, and for finding certain artifacts in the code.

Respecting coding conventions and design styles. Many virtual classifications are based on coding conventions and design styles used by the programmers. A problem with such classifications is that their precision depends on how well these conventions and styles are respected in the implementation. A partial solution to this problem is to enhance the development environment with support for using and enforcing such conventions and styles.

Defining virtual classifications is not trivial. A software architect explicitly has to declare logic predicates that classify implementation artifacts in conceptual groups. This complex task could be simplified in various ways. First of all, a number of predefined auxiliary predicates that are often used to define virtual classifications can be provided. It is even possible to define template predicates that capture typical patterns of classification. (This was exactly the purpose of our declarative framework.) Secondly, a tool like the Classification Browser (see Subsection 2.3.3 and [12]), which already supports the definition of manual classifications, could be extended to support the definition and manipulation of virtual classifications as well. Such a tool might provide a user-friendly interface for transparently constructing virtual classifications.

Defining virtual dependencies is at least as complex as defining virtual classifications and can be facilitated in similar ways: by providing predefined virtual dependencies; by providing a predefined set of auxiliary predicates for defining virtual dependencies; by providing parameterized predicates that capture commonalities among virtual dependencies; by providing a special tool that allows us to transparently construct virtual dependencies without explicitly having to write them as logic predicates; etc.

Reverse engineering architectural views from the implementation is not so easy and could be supported by tools that (semi-)automatically extract such architectural views from the implementation.

8.3.2 Re-engineering the software

Documenting a software system by means of a software architecture does not make the system more stable. However, it may improve the software understanding and enable the detection and correction of mismatches with the desired architecture when changes are made to the software. Furthermore, the architecture can make clear some of the imperfections of the software system, and allows us to assess some of the system qualities. It is important to invest in re-engineering the software and its architecture, so that some of its problems are fixed and its qualities are enhanced. Such a re-engineering step *can* make the software system more stable.

Amongst others, the following tasks in the re-engineering process could be supported by tools:

Understanding and browsing the implementation. As for reverse engineering, sophisticated tools for browsing and navigating through the implementation, and for finding certain implementation artifacts, can be of great help when re-engineering the software.

Architecture-driven browsing. Because the re-engineering process is driven by the software architecture, tools are needed that allow us to browse the implementation from an architectural point of view. For example, browsing all implementation artifacts that correspond to a certain architectural concept, navigating through the implementation based on some architectural relation, computing all artifacts that are in a certain (architectural) relation with another artifact, etc. A tool like the Classification Browser [12] supports this kind of architecture-driven browsing to a certain extent.

Code-generation facilities could be used, when re-engineering the software, to (partially) generate code so that the implementation (better) conforms to the architecture.

Refactoring the implementation is an important part of the re-engineering process that could be supported by tools such as the Refactoring Browser [61, 70, 69].

Architectural deviations are changes to the source code that do not conform to the architecture. Although such changes are not allowed, sometimes they cannot be avoided. To deal with such changes, support should be offered so that the deviations are explicitly documented, and so that the conformance checking algorithm can take them into account. We already mentioned this as future work in Subsection 6.4.4.

Resolving conformance conflicts. When the implementation no longer conforms to the architecture, either because the implementation or the architecture has evolved, we need support for resolving the conformance conflicts. We already explained in Subsection 7.4 how we extended our conformance checking algorithm so that it can produce more detailed information on which particular architectural relation is violated and on what are the implementation-level artifacts that caused the violation.

Incremental conformance checking is useful to assess the impact of small changes to either the implementation or to the architecture, without re-checking conformance entirely. Depending on the changes that were made, the incremental algorithm decides which parts may require re-checking. (See Section 8.1.)

Partial conformance checking is related to incremental conformance checking, except that it is not the conformance checking algorithm, but the architect who decides for which parts of the architecture conformance should be checked. During the re-engineering process, it cannot always be guaranteed that the implementation conforms to the architecture. However, even in those cases we may still want to check architectural conformance partially, or for incomplete architectures. For example, by checking conformance for some designated architectural relations only, or only for those architectural concepts and relations for which an architectural mapping has been defined.

Again, we refer to Subsection 8.3.4 for a discussion on some tools that may facilitate the above activities.

8.3.3 Synchronizing implementation and architecture

Once the implementation has been re-engineered in such a way that it has a 'clean' architecture, it is important to ensure that the software conforms to its architecture, whenever the implementation or architecture is modified. Obviously, an incremental conformance checking, which re-checks architectural conformance incrementally, can be of help here.

To achieve a real synchronization between an implementation and its conceptual architecture, however, the (incremental) conformance checking algorithm should be triggered automatically by the development environment whenever a change is made to either the implementation or the conceptual architecture. If conformance is violated, this should be reported to the developer or the architect, so that the conformance conflict can be resolved.

By enhancing the incremental conformance checking algorithm with such a synchronization mechanism, we can provide real support for co-evolution (see 2.2.2) of the implementation and its architectural views.

8.3.4 Tool support

In this subsection we describe some tools that support some of the activities that were enumerated in the previous subsections.

Partial conformance checking

In addition to an incremental conformance checker (as explained in Section 8.1), it could be useful to have a *partial conformance checker* as well. The purpose of the latter is to allow an architect to perform a partial conformance check of the implementation to an architectural view. To some extent, partial conformance checking is related to incremental conformance checking, in that the latter sometimes resorts to a partial re-checking of the conformance mapping. The main difference, however, is that with partial conformance checking it is the architect, and not the incremental conformance checking algorithm, who decides what should be re-checked. Some useful examples of partial re-checks could be:

- Checking conformance of some implementation module(s) only.
- Checking conformance to certain architectural relations only.
- Checking conformance to all architectural relations that are linked to some designated architectural concept.
- Checking conformance to incomplete architectures, i.e., only for the instantiated architectural concepts and relations.

Note that, per definition, our current conformance checking algorithm already supports partial conformance checking to a given architectural relation, as a full conformance check is defined as the conjunction of conformance checks for all architectural relations.

Architectural visualization tools

To address the need for representing architectural views graphically, one of our graduate students, J. Vanhentenryk, implemented a tool in which architectures can be drawn graphically, and which automatically generates the corresponding logic declarations. Most ADL toolsets include such graphical tools for visualizing and manipulating architectures, facilities for storing architectures, and certain domain-independent forms of analysis (such as checking for cycles or the existence of dangling connections) [27].

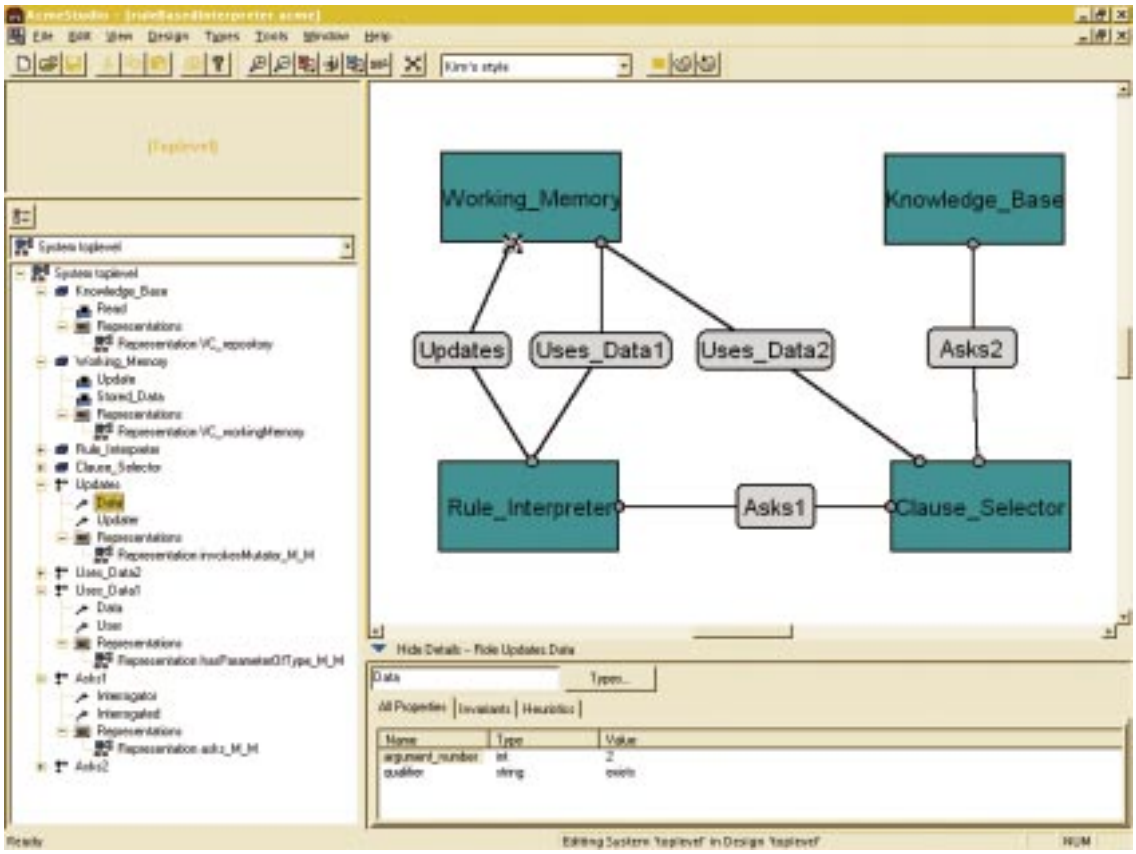

Figure 8.3: Visualizing the 'rule-based interpreter' view in AcmeStudio.

As an illustration, Figure 8.3 shows a graphical rendering of (part of) our 'rule-based interpreter' architectural view in AcmeStudio. AcmeStudio is an application for graphically editing architectural descriptions in the Acme ADL [27]. AcmeStudio includes a mechanism to create different 'diagram styles', which define the visualizations for the different types of architectural entities. By using this mechanism, a user can use his or her own customized graphical notations (domain-specific notations, user-defined notations, ...). In fact, Figure 8.3 illustrates how we could use AcmeStudio to represent an architecture in our own particular notation.

Customizable graphical representations can be used for many purposes. In general, customized notations may enhance understandability, for example, by using notations that are specifically tuned towards a particular architectural style (e.g., 'pipe and filter'). In one and the same architecture, we can even use different notations for different concepts (or relations), depending on their type; for example, in Figure 6.1 we used cylinders for concepts representing data and rectangles for concepts representing code. Finally, a simplified notation could be used to hide details, for example, by representing architectural relations as simple (labeled) arrows, instead of representing them as rounded rectangles with links connected to their roles.

Sophisticated browsers and finders

To find our way through the implementation, and to recover architectural and design abstractions from the implementation, we need sophisticated source-code browsers and finders.

- *Smalltalk browsers.* An environment like *VisualWorks*^{TM} Smalltalk already provides some interesting browsers for browsing class hierarchies, finding senders and implementors of methods, and so on.

- *The Classification Browser* significantly enhances Smalltalk's capabilities of navigating through the code. First of all, it includes some more advanced browsing facilities, to make it easier to recover software classifications from the source code. Furthermore, these recovered software classifications enhance the insight in the implementation by providing alternative views on the software (especially when cross-cutting classifications and multiple classification of implementation artifacts are allowed), and can be used in other source-code searches (for example, to restrict the scope of a search to those artifacts that belong to some classification of interest).

- *The Structural Find Application* (see Subsection 4.2.1) is a powerful source-code finder that facilitates the construction of complex search queries, such as finding the "class with name matching pattern `SOUL*App` and with a method with name matching pattern `showResult:` and with a method that sends `interpretIn:`".

- *The Query Application* (also see Subsection 4.2.1) is even more powerful, as it provides the user the full expressive power of a LMP language for querying the implementation of a software system. The same kind of querying can also be done in our Prolog setup. During our manual reverse engineering of architectural views from the SOUL implementation, we often used our LMP language precisely for this purpose. Many of these queries eventually found their way in the definition of various virtual classifications and virtual dependencies.

Enforcing coding conventions and design styles

As explained on pages 70 and 134, the use of coding conventions (and styles) in virtual classifications may lead to problems when the conventions are not followed. It is important to see things in the right perspective, though. In a language like Smalltalk, it is common practice to use and respect certain naming and coding conventions (see, for example, K. Beck's book on Smalltalk best-practice patterns [5]). This self-inflicted discipline of Smalltalk programmers is a kind of counter-measure for the fact that a dynamically typed language like Smalltalk often provides too much flexibility. Because of this discipline, defining virtual classifications that rely on conventions may not be as dangerous as may seem at first sight. In typed languages, conventions are typically less respected, but there is also less need to define virtual classifications in terms of conventions: thanks to the extra type information, virtual classifications can often be defined more precisely without using conventions.

Nevertheless, it still remains possible that virtual classifications are defined in terms of coding conventions, and that problems arise when these conventions are not respected. To avoid such problems, it would be useful if the development environment would provide support for using and enforcing such conventions. Support for conventions and styles includes: *checking* whether some implementation artifacts follow a certain convention, *detecting* implementation artifacts that do not follow the convention, *enforcing* that certain conventions are respected in the implementation, *generating* code templates for implementation artifacts so that they automatically follow some convention, *transforming* unconventional code to code that does conform to the conventions, and so on.

The SOUL-Smalltalk combination has proven to be an ideal medium for building sophisticated software engineering tools that provide the above kinds of support. We repeat from Section 2.2 that experiments have already been carried out to support best-practice patterns, idioms, and

coding conventions [54]; to detect and check design patterns in Smalltalk source code [86]; to log violations of certain programming conventions and styles in a 'to do' list dynamically; and more recently to generate code that conforms to some convention (for example, automatically generate the accessor methods for all instance variables of a class).

Predefined predicates

Both virtual classifications and virtual dependencies are defined as logic predicates that can make full use of the power of LMP. Although this has many advantages from the viewpoint of expressiveness, it does require a lot of insight from the architect in both the implementation and the LMP language. To simplify the declaration of such virtual classifications and virtual dependencies, and to avoid that an architect should re-invent the same predicates over and over again, the conformance checking tool provides a whole range of predefined predicates in terms of which the architect can define his or her own logic predicates. As explained in Subsections 5.3.3 to 5.3.6, these predicates are defined in the DFW. We refer to those subsections for a more elaborate discussion of this library of predicates.

Classification browser

The Classification Browser, first discussed in Subsection 2.3.3 can be used in many different ways. We already mentioned some of its powerful navigation capabilities and its ability to manipulate manually-defined software classifications.

A tool like the Classification Browser could be extended with support for defining and managing *virtual classifications* as well. Such a tool can provide a user-friendly interface for transparently constructing virtual classifications. One way of achieving this is for the tool to provide access to the set of predefined and template predicates of the DFW, and to support the interactive construction of more complex predicates using logic operators. Another way is related to how macros are recorded in, for example, MS-Word. We would use the tool for browsing the implementation using its advanced navigation facilities, while in background the tool records all actions that are undertaken by the user. Afterwards, if the user decides that the correct classification has been constructed, the tool can restore the set of performed actions. These actions form the description of the constructed virtual classification.

Later, we will also explain how a combination of the Classification Browser with a software tagging mechanism can be used, to a certain extent, for architectural recovery.

Dependency browser

Similar to the proposed extension of the Classification Browser with support for virtual classifications, it would be useful to have a kind of *Dependency Browser* as well. This Dependency Browser should support the construction of virtual dependencies, and provide some powerful navigation capabilities based on these virtual dependencies. Using a similar approach as for the extended Classification Browser it could provide a user-friendly interface for transparently constructing virtual dependencies, without requiring a user to explicitly write them as logic predicates.

Architectural extraction

We already discussed how manual reverse engineering of software architectures could be supported. Obviously, we would also like some support for (semi-)automatically extracting architectures from the implementation. We mention three relevant techniques:

- *Ontologies.* D. Deridder and B. Wouters make a case for the application of *ontologies* in the domain of software engineering [16]. They state that by integrating techniques and formalisms from the domains of computer linguistics and artificial intelligence (in particular, ontologies and ontology-related techniques) in existing software engineering tools, the software development process may be enhanced significantly. An ontology-based experiment was

conducted to reverse engineer UML diagrams from an existing application. It should be investigated whether a similar experiment could be set up to reverse engineer the architecture of an existing system.

- *Architectural recovery through software tagging.* K. De Hondt and P. Steyaert confirm that a fundamental problem with large evolving software systems is a bad understanding of the software architecture [13]. They claim that there are two crucial ingredients in managing software evolution: the ability to trace past activities (i.e., keeping track of the changes that were made, as well as the reasons why these changes were made) and the ability to capture emergent patterns (i.e., capturing the architecture, components, and object collaborations that were not recognized at the start of a project and that emerge as a result of development activities). *Software classification* (see Section 2.3) is proposed as a general framework that provides the ability to trace past activities and to capture emergent patterns. They support the use and definition of software classification by means of two tools: *software tagging* and the *Classification Browser*. We already discussed the Classification Browser on page 177.

  The idea of *software tagging* is that when software engineers carry out development tasks, they usually know the context in which changes are made. They know the module they are changing, the software layer a class belongs to, the specification they are implementing, the bug they are fixing, etc. Normally, this knowledge is kept implicit in the heads of the software developers. Software tagging makes this knowledge explicit, by requiring the software engineers to transfer that knowledge in the form of classification information when changes are made, and registering that information in the form of a tag in the software. Examples of tags are: time of change, modifier, activity that gave rise to the change, customer for whom the change was made, module, task (i.e. new development, implementation of a specification, bug fix, code review, testing, etc.), intention. Software tagging provides important information on a software system, which can be used to define some interesting software classifications.

  Furthermore, K. De Hondt and P. Steyaert explain how software classification may be used for the purpose of *architectural recovery* [13]. Whereas architectural concepts often exist only as conceptual entities in the heads of the developers, when performing a development task the developer needs a physical view in terms of classes and methods. By tagging the different classes and methods with the concept they belong to, a mapping is obtained of architectural concepts to the relevant classes and methods. The recovery of architectural concepts with classification through software tagging results in multiple architectural views on software. By browsing the generated classifications, the developers get a picture of the software in terms of architectural concepts. The concepts that live in their heads have now become physical entities (i.e. classifications) in the software development environment.

- *Computing divergences.* Another kind of information that could be useful to extract from an implementation is where it *diverges* from its architecture. Divergences are important dependencies that are present in the implementation, but are not reflected in the architecture. Such information could be useful, for example, during the re-engineering process: probably it is a good idea to update either the architecture to explicitly include the important divergences, or to modify the implementation to get rid of these divergences.

  In G. Murphy's approach to architectural conformance checking [57], she computes the *convergences* (where the implementation agrees with the architecture), the *absences* (where the implementation does *not* contain dependencies that are described by the architecture) and the *divergences* (where the implementation has dependencies that are not predicted by the architecture). Our conformance checking algorithm computes the convergences and absences only. If conformance checking succeeds, there are no absences, only convergences. If conformance checking fails, this is caused by absences of expected architectural relations in the implementation. Our approach does not locate divergences, however.

In our approach, one architectural view does not provide a 'complete' picture of a software implementation. Typically, there are multiple architectural views that all contribute to the system's architecture. Hence, although the implementation of the software system may contain dependencies that are not described by some architectural view (i.e., divergences), these dependencies may be described by another architectural view. So if we want to compute the divergences in our approach, we need to specify clearly with respect to which view(s) of the conceptual architecture the divergences should be computed. For example, if we do assume that some set of architectural views is supposed to provide a 'complete' picture of the system's architecture, it can be useful to generate the divergences with respect to these views. Since the picture is supposedly complete, there should be no implementation dependencies that are not reflected in this (set of) architectural views.

There is a more important difficulty with computing the differences, though. In our approach, computing the divergences will be an extremely computationally-intensive process, because of the vast amount of implementation dependencies possible, and the even larger amount of ways in which these may be combined and abstracted into architectural relations. G. Murphy did not have this problem, because her approach always works with a restricted set of relationships. She did not compare the full source code to the architecture, but only a much smaller 'source-code model' (e.g., a call graph), which was extracted from the source code by some tool. Therefore, in practice, to be able to compute the divergences in our approach, we should restrict the scope to a certain set of implementation dependencies or architectural relationships.

Refactoring Browser

The Refactoring Browser [69, 70] is a tool for restructuring the implementation of an object-oriented software system in a behavior-preserving way. A typical example of a 'refactoring' is the following: suppose we have some superclass of which all direct subclasses introduce the same variable, which is not present in the superclass itself. In that case this variable can be removed from all subclasses and added to the superclass.

Although the Refactoring Browser is not an 'architectural tool', it can be useful during the re-engineering phase. Refactorings are typically used to 'clean up' a software system, which may result, for example, in a cleaner architecture as well.

Code generation

T. Tourwé and K. De Volder discuss how software classifications can be used to drive code generation [82]. The general idea behind their proposal is that a classification groups together several related entities which share some characteristics. Often, this is reflected by these entities having in common some state and behavior. Since these entities need not in any way be related through inheritance, the state and behavior is spread out and duplicated. When using code generation on classifications, this problem can easily be alleviated. Instead of manually duplicating the code over the different entities, we can define behavior and state on the classification itself and let the code generator take care of all the work.

As a simple illustration of their approach, they give the example of the Visitor design pattern. When implementing this pattern, there is a lot of code duplication in the different classes that are visited.[2] Now suppose that we define a virtual classification consisting of all visited classes. Instead of writing the duplicate code again and again for each such class, it is more opportune to define the duplicate code on the virtual classification, and use a code generator to automatically generate the correct code necessary for all classes in the classification.

[2] For the sake of the argument, we simplified the example somewhat. In fact, the code duplication depends not only on the visited classes, but on a combination of the visitor and the visited classes. We refer to [82] for the full example.

As architectural concepts precisely correspond to virtual classifications of implementation artifacts that may be spread over the entire implementation, we have exactly the situation sketched above. The proposed code-generation approach can, for example, be useful when re-engineering a software system so that it conforms to a new architecture. Suppose that we add an architectural relation between two architectural concepts in some architectural view. This simple architectural evolution may have a high impact on the implementation. For example, it might involve the addition of an extra invocation relationship from every artifact corresponding to the first architectural concept to some artifact corresponding to the second concept. With a code generation approach, however, the implementation changes that are required may be defined on the architectural concepts themselves, and be carried out automatically by the code generator.

To support this kind of code generation based on virtual software classifications, T. Tourwé and K. De Volder suggest using a *Codification Browser* which provides a much more intuitive and user-friendly interface, than when everything needs to be specified at the level of a LMP language.

Synchronization

There is a close relationship between the research in this Ph.D. dissertation and R. Wuyts' Ph.D. research [87]. However, whereas the focus of our research is to develop an architectural formalism for automated conformance checking of implementation to architecture, the focus of Wuyts' research is on techniques for synchronizing design and implementation of a software system. Wuyts also adopts a LMP approach, and argues that in a LMP medium, *synchronization* between design and implementation can be achieved in a variety of ways:

- *Conformance checking* corresponds to evaluating a query which verifies whether the implementation conforms to the design.
- *Enforcement* expresses design as a constraint on the implementation, and generates a warning when the implementation violates the rules describing the design.
- *Generation* assures synchronization by generating parts of the implementation from the design, or vice versa.

These different approaches towards synchronization have varying degrees of 'strongness'. Conformance checking is a rather weak kind of synchronization. The check is initiated by a software engineer, and only immediately after the conformance check the engineer knows whether the implementation conforms to the design. Enforcement is a stronger kind of synchronization, although the 'strongness' depends on how and when violations are reported. For example, a weak form of enforcement could check for violations in background, and log them in a to-do list for later inspection by the software engineer. A strong form of enforcement could check for violations whenever a change is made to the system, and provide immediate support to the software engineer to resolve potential conflicts interactively. Finally, generation is a very strong form of synchronization, in the sense that the generated code (resp. design) will be conform to the design (resp. code) by construction. However, it is weak in the sense that the generation is typically initiated by a software engineer, and that the code is only in conformance with the design immediately after generation has taken place.

It is outside the scope of this dissertation to investigate which synchronization technique is most opportune or how these techniques can be implemented. This is the subject of Wuyts' Ph.D. dissertation [87]. Whereas Wuyts' contribution lies in providing a framework and environment that can handle several forms of synchronization (i.e., conformance checking, generation and enforcement), our contribution lies in providing an architecture language in which to describe software architectures (the ADL) and their mapping to the implementation (the AML). The only overlap is that we both adopt a LMP approach and that we both support conformance checking. Our dissertation builds on this conformance checking to provide full support to describe and check conformance to architectures, but 'neglects' other forms of synchronization. Wuyts focuses on supporting different forms of synchronization, but only provides ad-hoc support for some design notations.

8.3.5 Conclusion

The goal of this section was to investigate how the conformance checking tool proposed in this dissertation could be enhanced to an industrial-strength environment for architectural-driven software development. Because of the many tools and techniques we mentioned, it may seem that we still have a long way to go before such an environment can be constructed. It should be stressed, however, that most of the required techniques and tools already exist or are currently under investigation. Moreover, Smalltalk prototypes of most of the discussed tools are available. Therefore, it is not unrealistic to assume that such a state-of-the-art environment can actually be constructed by combining all these tools in a Smalltalk setting. More specifically, our state-of-the-art industrial-strength environment could consist of a mixture of:

- a *VisualWorks*^{TM}-like development environment, enhanced with
  - sophisticated browsers, finders and source-code navigation facilities
  - an enhanced classification browser and editor (including support for virtual classifications and virtual dependencies)
  - support to enforce coding conventions and styles
  - a refactoring browser
  - an ADL with a customizable graphical user interface
- a Prolog-like LMP language with a tight symbiosis with all of the previous. The logic language comes with
  - a predefined library of logic predicates for defining virtual classifications, virtual dependencies, etc.
  - an architectural conformance checker (including support for synchronization of the implementation and the architecture)
  - an architectural extractor
  - code-generation facilities

8.4 Generalizing the formalism

The case study we performed in Chapter 7 focused on checking conformance of some Smalltalk implementation (i.e., SOUL) to some architectural views. However, the proposed formalism is sufficiently general to allow architectural conformance checking of implementations written in other *object-oriented* programming languages (e.g., Java) or in entirely *different programming paradigms* (e.g., a logic programming language), or even conformance checking of design diagrams (e.g., UML class diagrams). Although we did not implement nor experiment with any of these generalizations, in this section we discuss how they could be achieved, thus illustrating the generality and expressiveness of the formalism even more.

8.4.1 Other object-oriented languages

When porting the approach to an object-oriented language other than Smalltalk, obviously many changes are required to the predefined predicates of the DFW. Additional predicates are needed for manipulating those language constructs that are not present in the Smalltalk language.[3] Also, we need to re-implement the predicates for those language constructs that have a different semantics than their Smalltalk counterparts (e.g., Smalltalk does not support multiple inheritance). For those language constructs that only underwent a change in syntax, however, we have good faith that the necessary changes can be kept within reasonable proportions, for several reasons.

1. The DFW is implemented as a layered library of rules. If we disregard the technical logic meta-programming layers of the DFW, the lowest layer is a Smalltalk-specific layer which defines some primitive predicates for manipulating Smalltalk source-code artifacts and implementation relationships. On top of this 'representational' layer resides a 'base' layer that adds some structural predicates defined directly in terms of the more primitive predicates. Higher-level layers describe more high-level relationships, such as coding conventions, design patterns and predefined architectural mapping schemes. Ideally, when switching to another language, we only need to change predicates in the lowest layer(s). (In practice, however, some higher-level predicates may require some changes as well.)

2. As most object-oriented languages have similar language constructs (e.g., message sends, assignments, return statements) the parse-tree representations of methods will show many similarities. Of course, as other object-oriented languages contain additional language constructs, the parse-tree structure will show some differences as well. But because our predicates typically do not manipulate parse trees directly, but use a high-level parse-tree traversal predicate instead, these differences may remain hidden for most predicates. Ideally, the only thing that is required for those predicates is a re-implementation of the parse-tree traversal predicate.

3. The implementation artifacts are stored in the source-code repository in a format that is fairly general and language-independent. (Even the method parse-tree representations have essentially the same format, although they may contain some language-dependent constructs.)

Of course, it remains to be investigated in practice whether the port to a new language will be as easy as suggested above. But even if we would have to change all predicates, we can find comfort in the fact that it only needs to be done once for that particular language. The important thing is that after the DFW has been re-implemented for the new language, the conformance checking tool can be used immediately.

The only thing that remains to be done is to make the implementation artifacts for some software system in the new language available to our conformance checking tool. As explained in Subsection 6.1.3, all implementation artifacts are stored in an external repository. Therefore, we

[3] Smalltalk has a fairly small and clean syntax. Most other languages contain additional constructs that are not provided by the Smalltalk language. Obviously, all predicates for reasoning about these constructs will need to be implemented from scratch.

merely need to provide a new ODBC-compliant repository containing implementation artifacts in the new language. This can be done in two ways:

1. Store the implementation artifacts in a database with the same format as the database in which our Smalltalk implementation artifacts were stored. This is possible, because we used a language-independent format, that can be used for representing either Smalltalk source code, Java source code or even UML class diagrams [51].

2. Use an existing source-code repository with a different format, and redefine the Prolog-predicates that implement the repository-access layer of the DFW.

One main difference between Smalltalk and many other object-oriented languages is that Smalltalk is dynamically typed. In statically typed languages, we can make use of the static type information to define high-level implementation relationships statically. In Smalltalk, due to the lack of static type information, we often encountered problems when we tried to define complex implementation relationships statically. To circumvent these problems we implemented some (computationally-intensive) predicates that infer the type of certain expressions. Also, we often relied on naming and coding conventions, taking advantage of the fact that Smalltalk has a rich 'culture' containing many conventions and 'best practices' that are used by most Smalltalk developers.

8.4.2 Design diagrams

A lot of contemporary CASE tools provide good support for mapping design to source code, as design is quite close to the code.[4] But there is not yet a good mapping from architecture to design. In such a context, it might be more relevant to apply our architectural conformance checking approach to design artifacts than to implementation artifacts.

Nothing prohibits our approach to be used for checking conformance of the *design* of a software system to its architectural views. The formalism does not require any changes, except that the LMP language will need to reason about design artifacts instead of about implementation artifacts. As mentioned in the previous section and in Subsection 6.1.3, the same database format we used for storing Smalltalk implementation artifacts, can be used to store UML class diagrams. In the context of an industrial research project [51], a generator was developed which can generate a database containing this information from a CASE tool like *Select Enterprise*^{TM}. To manipulate and to reason about the data in this database, to a certain extent we can make use of the same primitive Prolog predicates as we used for reasoning about Smalltalk implementation artifacts. Of course, as in the previous generalization we will also need to add some new predicates and change some existing predicates.

8.4.3 Logic programming language

Taking the Prolog implementation of our prototype conformance checking tool as an example, we now explain how our approach could be generalized to support architectural conformance checking of a Prolog implementation. In fact, this generalization is fairly straightforward. The only thing we need is a LMP language that can reason about Prolog implementation artifacts instead of about Smalltalk implementation artifacts. As Prolog is a reflective language, we simply choose Prolog both as LMP language and as base language. We do not need to change anything to our implementation of the conformance checking formalism, except for the primitive predicates that define the mapping of the base language to the meta language. Instead of mapping Smalltalk implementation artifacts (e.g., classes, methods) to Prolog terms (e.g., `class('SOULTerms',1989)`, `method('at:',1992)`) and implementation dependencies (e.g., method invocation, class instantiation, inheritance) to predicates (e.g., `invokes_M_M`, `createsInstanceOf_M_C`, `specializes_C_C`),

[4] For example, our industrial partners at Getronics [51] follow a development approach whereby a large part of the source code is generated automatically from UML design models.

the mapping now becomes a reflective mapping. Prolog implementation artifacts (i.e., predicates, rules, facts, queries and files) and dependencies (e.g., Prolog calls) are mapped to Prolog terms and predicates, respectively.

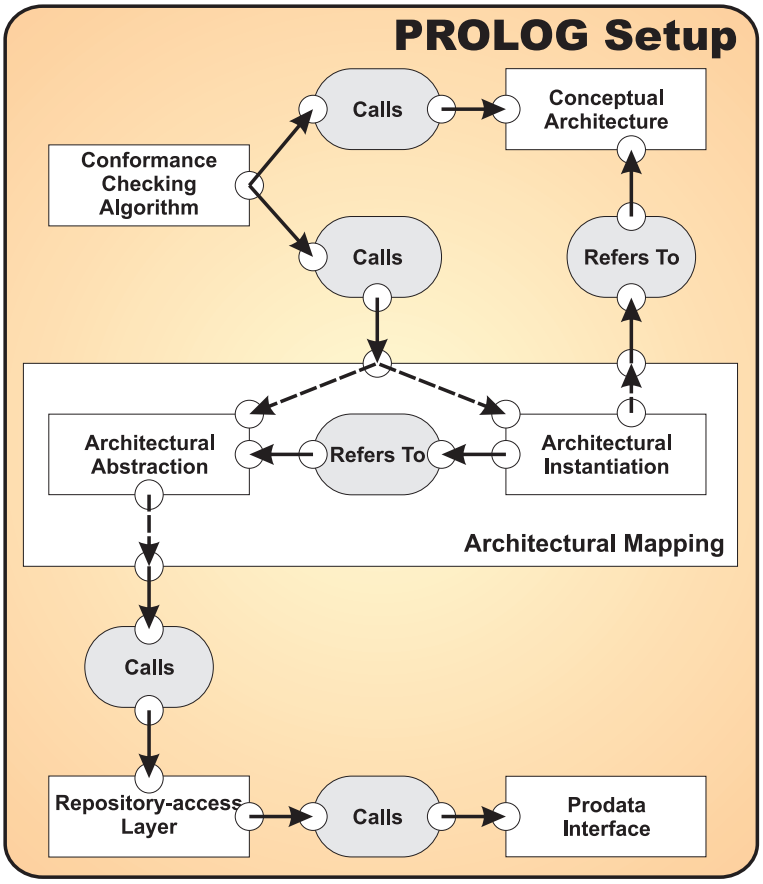

Figure 8.4: An architectural view describing the Prolog implementation of our conformance checking tool.

Let us illustrate this with a concrete example. The architectural view of Figure 8.4 describes the structure of the Prolog implementation of our prototype conformance checking tool. Apart from a difference in notation, this figure is essentially a retake of the top part of Figure 6.3 on page 100. Note that we left out those parts that are not implemented in Prolog, such as the ODBC access to an external source-code repository. All architectural concepts in Figure 8.4 correspond to some part(s) of our Prolog implementation. Table 8.1 mentions how precisely each of the architectural concepts in the architectural view of Figure 8.4 can be mapped to the Prolog implementation. The first column mentions the name of the concept, the second column indicates the kind of Prolog artifacts to which this concept is mapped, and the third enumerates the names of these Prolog artifacts using '*' as a wildcard symbol.

As can be seen from Table 8.1, the concepts in the architectural view are typically mapped to sets of files or predicates. We did not describe these sets by explicitly enumerating all different files or predicates. Instead, we provided a *virtual* description using a wildcard to describe the names of all files or predicates in some set. For example, all predicates in the *Prodata*^{TM} interface start with the string 'db_' (e.g., db_add_record, db_sql, db_commit, etc.) [42], and all files declaring facts of the architectural instantiation start with the string 'arch_mapping_' (e.g., arch_mapping_soul_implementation.pl, arch_mapping_soul_userinteraction.pl, etc.). As before, we can define the architectural mapping for architectural concepts in terms of virtual classifications.

Concept	Artifact kind	Prolog artifact(s)
Conformance Checking Algorithm	set of files	`arch_conformance_checking.pl` `arch_adl.pl`
Conceptual Architecture	set of files	`arch_architecture_*.pl`
Architectural Abstraction	set of files	`arch_vc_*.pl` `arch_implementation_relations*.pl`
Architectural Instantiation	set of files	`arch_mapping_*.pl`
Repository-access Layer	set of files	`arch_repository_*.pl` `arch_parsetreetraversal_st.pl`
Prodata Interface	set of predicates	`db_*`

Table 8.1: Mapping architectural concepts to Prolog artifacts.

For example, the following rule describes the virtual classification for the **Architectural Instantiation** concept.

```
classifiedAs(file('ArchitecturalInstantiation'), File) :-
    prologFile(File),
    fileName(File, FileName),
    patternMatch(FileName, and(prefix('arch_mapping_'), postfix('.pl'))).
```

Note that most of the virtual descriptions in Table 8.1 are based on certain naming conventions. As in the Smalltalk case, Prolog programmers tend to use (and respect) a lot of these conventions, to counter the fact that Prolog is an untyped language with little structuring facilities.

The architectural view of Figure 8.4 contains two kinds of architectural relations: *Calls* and *Refers To*. The *Calls* relation is defined in terms of a virtual dependency which checks for an ordinary calling relationship between Prolog-predicates. For example, the **Repository-access Layer** contains some predicates (e.g., `addRecord`) that *call* predicates from the **Prodata Interface** (e.g., db_add_record):

```
addRecord(PredicateName, Record) :-
    table(PredicateName, TableName, AccessType),
    db_add_record(TableName, Record),
    ( AccessType = load ->  createTupleFact(PredicateName, Record);
      otherwise         ->  true ).
```

In addition to calling another predicate, a predicate can also *refer to* some other predicate so that this other predicate can be called dynamically later. A first example of this kind of relation was given in Subsection 6.2.3 where we explained how the **Architectural Instantiation** could be declared, by means of facts like the following:

```
conceptMapping(soulUserInteraction, inputWindow, userInput).
```

This particular fact *refers to* both an element of the **Conceptual Architecture**, namely the concept '`inputWindow`', and an element of the **Architectural Abstraction**, namely the virtual classification '`userInput`'. This information is used later during conformance checking. (For example, when checking conformance to the `inputWindow` concept, the `userInput` classification will be computed by calling the appropriate predicate.)

The virtual dependencies that codify the *calling* and *refers to* relationships reason about single predicates. We can apply them to files as well, by considering a file as a set of predicates. For example, a file *calls* another file when one of its predicates *calls* one of the predicates in the other file.

To conclude this section, we mention that some Prolog versions provide more reflective capabilities than others. It should be investigated whether there exists a Prolog version with enough reflective power to reason about Prolog source code at a sufficiently fine-grained level. For example, we should be able to reason about which predicates belong to a certain file; about which

predicates (predefined as well as user-defined) are available in the Prolog system; about which predicates are called or mentioned in the body of some predicate, rule or fact; and so on. If such a Prolog version would not be available, we can still conduct an experiment using SOUL as a logic language. SOUL has the advantage of having a completely open implementation, so that we can add whatever reflective capabilities that are needed.

8.4.4 Other programming languages

The previous subsection illustrated how our architectural conformance checking approach could be applied to a logic programming language like Prolog. In fact, our approach could be generalized to any other programming language as well. The only thing that is needed is a LMP language that can reason (at a sufficiently fine-grained level) about artifacts and dependencies in the programming language of interest. This can be achieved either by implementing a reflective logic meta layer in that language (as in the SOUL setup), or by providing an external source-code repository that can be accessed from within Prolog and by defining the necessary Prolog predicates for accessing the source code in that repository (as in the Prolog setup). In addition, we need to define a DFW of predefined logic predicates for reasoning about and manipulating artifacts and dependencies in that language.

8.5 Summary

In this chapter, we elaborated on some future work that is required to elevate our prototype architectural conformance checking tool to an 'industrial-strength' tool. We first mentioned some interesting optimizations, including a more incremental version of our conformance checking algorithm. Then we talked about how our tool could assist an architecture-driven development approach, and which other tools would be useful or needed to support such an approach. Finally, we discussed how to generalize our architectural conformance checking approach to reason not only about object-oriented implementations, but also about design models or implementations in other programming languages.

Chapter 9

Conclusion

Throughout this dissertation, we defended the thesis that automated support for checking conformance of the implementation of a software system to its architectural views, can be achieved in a very expressive way by adopting a logic meta-programming approach. We summarize how we supported and validated this thesis, and repeat why it is a solid advancement to the field. We conclude with an enumeration of the major contributions of this dissertation, and mention some future research topics.

9.1 Summary

Architectural conformance checking is the task of verifying whether the implementation of a software system corresponds to the more high-level structure described by its software architecture. In the introduction of this dissertation, the following thesis was put forward:

> Automated support for checking conformance of an implementation of a software system to its architectural views can be achieved in a very expressive way by adopting a logic meta-programming approach.

To reduce the scope of the thesis, we confined ourselves to static conformance checking. In other words, we only reason about the static structure of a software implementation, and do not take dynamic (i.e., run-time) information into account. Another restriction we made was to consider only object-oriented implementations, and Smalltalk implementations in particular.

We supported the thesis by presenting an elegant and simple architectural formalism, together with an algorithm, for automatically checking conformance of the implementation of a software system to one or more architectural views. To verify the feasibility of this formalism and algorithm, a prototype of a conformance checking tool was implemented. Using this prototype, a case study was conducted on an existing medium-sized Smalltalk application consisting of about 100 classes. Based on the results of this case study, we suggested some future improvements of, and optimizations for, our conformance checking formalism and tool. One interesting extension was explained in more detail: an incremental version of the conformance checking algorithm. This extension is particularly useful in the context of evolution (of either the implementation or the architecture). As further evidence of the generality and expressiveness of our approach to automated architectural conformance checking, we hinted on how it could be generalized to other programming languages (not necessarily object-oriented), or even design languages.

The use of a LMP approach was a crucial and deliberate decision in the design of our architectural conformance checking approach. As most existing architectural conformance checking approaches lack expressiveness, we decided to offer the full power of a meta language to the architect, thus providing him or her with a maximum of flexibility in defining the mapping of architectural entities to implementation artifacts. A *logic* meta-programming approach was advocated because it enables an architect to describe this architectural mapping in a very expressive,

yet concise and intuitive, way. To prove (amongst others) the expressiveness of the proposed LMP approach, we presented a list of requirements that our conformance checking formalism should satisfy and showed, on the basis of our case study, how each of these requirements was satisfied in our formalism.

The same LMP language that was provided to the architect, was also chosen as implementation medium for constructing the prototype conformance checking tool. The main reason for this choice was that the proposed formalism itself has a strong logic flavor. We showed how the use of a LMP language allowed us to implement the conformance checking algorithm and architecture language in a very straightforward manner. Finally, we elaborated on how the prototype and formalism could be extended to obtain a realistic and practically usable tool that provides automated support for checking architectural conformance. In Section 9.4, we will discuss some more future work.

9.2 Conclusion

Software architectures are increasingly recognized as important design abstractions. They provide a simple mental picture that allows software engineers to grasp the global structure of a software system. Software architectures enhance the understandability of large and complex software systems and make it easier to maintain and modify these systems. Without support for checking conformance between the implementation of a system and the architecture, however, the implementation will quickly drift away from its architecture, thus loosing all beneficial properties of having an up-to-date architecture.

Checking conformance of an implementation to one or more architectural views is a non-trivial problem, especially when allowing architectural views that can cut across the implementation structure. Although some architectural conformance checking approaches exist, they all lack expressiveness. Typically, they impose some restrictions on the mapping of the architecture to the implementation. Either they restrict the kinds of implementation artifacts and/or dependencies that can be considered, or they disallow cross-cutting architectural mappings. In contrast, this dissertation proposed a more expressive architectural conformance checking approach based on LMP, which does not pose these restrictions. It allows an architect to declare complex architectural mappings in terms of virtual classifications and virtual dependencies which are expressed as logic predicates in a LMP language. Our case study illustrated that our consistent use of a LMP language throughout all abstraction layers — from the implementation level over the architectural abstraction and architectural instantiation to the conceptual architecture — provides a viable and expressive formalism to describe architectural knowledge at a sufficiently high level of abstraction while still allowing conformance checking of the implementation.

The architectural conformance checking formalism proposed in this dissertation is a first and important step towards solving the problems of architectural erosion and architectural drift. We provided a means of checking conformance of an implementation to its architecture, and even sketched an incremental conformance checking algorithm. To solve the problem completely, though, we need full support for co-evolution between architecture and implementation. More precisely, we still need an automated synchronization mechanism between the implementation and its architectural views, as well as support for simultaneous evolution of the implementation and its architectural views.

Finally, this dissertation confirms our beliefs that the emerging technique of LMP is an ideal medium in which to build state-of-the-art software engineering support tools. Subsection 2.2.1 already mentioned a list of such tools. This dissertation adds another one to that list.

9.3 Achievements

In this section, we elaborate on the artifacts that were produced in the context of this dissertation, and we repeat the main contributions.

9.3.1 Produced artifacts

The most important artifacts produced in the context of this dissertation were, of course, the conformance checking formalism and tool. The conformance checking formalism consisted of an architecture language and a conformance checking algorithm. We also discussed an incremental version of the algorithm but did not work it out in detail, nor did we implement it.

Two versions of the conformance checking tool were implemented. A first version was implemented in the SOUL language and used SOUL's close symbiosis with Smalltalk to reason about Smalltalk source code. A more recent version was implemented in Prolog and used ODBC to access implementation artifacts stored in an external repository. To generate a repository containing Smalltalk source-code artifacts, we implemented a 'database generator' in SOUL.

Another useful artifact was produced during our case study: we documented the architecture of the SOUL system in terms of three architectural views and their mapping to the implementation (see Chapters 4 and 7).

9.3.2 Contributions

We summarize the main contributions of this dissertation, in order of decreasing importance:

1. *A formalism and tool for architectural conformance checking.* We provided a general and expressive formalism and tool for automatically checking conformance of the implementation of some software system to its architectural views.

2. *The expressive power of logic meta programming.* We showed that the expressive power of LMP enables an architect to describe the architectural mapping in a very expressive, yet concise and intuitive, way.

3. *Virtual classifications.* We confirmed K. De Hondt's claim [12] that software classifications are a powerful means of capturing architectural abstractions in a software system. In addition, we promoted virtual software classifications as an even more expressive, elegant and intuitive way of representing architecturally relevant abstractions of implementation artifacts.

4. *Virtual dependencies.* Similar to the notion of virtual classifications, we illustrated how virtual dependencies constitute a high-level and intuitive mechanism for abstracting complex relationships among implementation artifacts.

5. *Multiple cross-cutting architectural views.* As an important side-contribution we illustrated the relevance of providing multiple overlapping architectural views. These architectural views may cut across the implementation structure.

6. *Logic meta programming as implementation medium.* We demonstrated that LMP is a suitable implementation medium for implementing the proposed conformance checking algorithm and architectural model.

7. *Incremental conformance checking.* We sketched how the original conformance checking algorithm could be refined into an incremental version. This incremental version has the advantage of being more efficient, in the sense that conformance only needs to be checked incrementally, depending on how the implementation or architecture has evolved.

Below, each of the above contributions will be discussed in a bit more detail.

Formalism for architectural conformance checking

The general formalism for architectural conformance checking we proposed has a layered structure. The four main layers are the conceptual architecture, the architectural instantiation, the architectural abstraction and the declarative framework. (The implementation could be considered as a fifth layer.) Furthermore, each of these layers itself has a layered structure. For example, the concepts and relations in the conceptual architecture can be described by sub-architectures consisting of other architectural concepts and relations. In the architectural abstraction, the virtual classifications and virtual dependencies can be defined in terms of other, more primitive virtual classifications and virtual dependencies. The declarative framework is a layered library of predicates, ranging from very high-level predicates that describe typical architectural mappings to very low-level predicates for reasoning about source code.

By combining the declarations in each of these layers, an algorithm for checking architectural conformance automatically can be constructed. More precisely, the mappings of architectural concepts to lower-level artifacts and of architectural relations to lower-level relations, are used to transform the high-level architectural relations among architectural concepts to verifiable predicates over implementation artifacts.

Although the formalism has been validated only for checking architectural conformance of object-oriented implementations, we are convinced it is general enough to support architectural conformance checking of other kinds of software artifacts as well (e.g., artifacts in other programming languages or in design languages).

We also sketched an incremental version of the conformance checking algorithm which does not re-check conformance entirely, when changes are made, but only re-checks those parts that are affected by the change. The incremental algorithm was based on a taxonomy of the different kinds of changes that can be made to the architecture, architectural mapping and implementation, and on an impact analysis of these changes on architectural conformance.

Logic meta programming

Just like K. De Volder [14] proposed to use LMP as a way to extend the expressiveness of current type systems (see 2.2.1), we proposed to use LMP to extend the expressiveness of current architectural conformance checking approaches. By defining the architectural mapping in terms of virtual classifications and virtual dependencies, which can make use of the full power of a LMP language, we obtained a very expressive conformance checking formalism. It allows an architect to declare complex architectural mappings in a reasonably intuitive and concise way.

We also used LMP to implement our architectural formalism. We repeat some of the reasons why LMP is a convenient implementation medium. A logic language is typically well suited for representing, describing and reasoning about (architectural) knowledge. LMP is also well-suited for meta programming and language processing. Furthermore, a logic language may be the most suitable implementation language, as the proposed formalism itself had a strong logic flavor. For example, the entire conformance checking algorithm revolves around the construction of a logical expression which can be evaluated to check for architectural conformance.

Virtual descriptions

The notion of virtual classifications plays a crucial role in the proposed formalism. The idea of using software classifications as an intermediary abstraction for describing architectural concepts is strongly inspired by K. De Hondt's work on architectural recovery in evolving object-oriented systems [12]. He promotes the use of software classifications as a powerful means of organizing implementation artifacts in a flexible and uniform manner. In particular, he uses these software classifications to capture architectural abstractions that were reverse engineered from implementation artifacts and their interrelationships.

Our case study confirms K. De Hondt's claim that software classifications offer an elegant and powerful abstraction mechanism for describing architectural concepts. By defining architectural

concepts in terms of software classifications, the details of the lower-level artifacts on which they are mapped are hidden, thus allowing us to reason about the concept's relationships with other architectural concepts independently of the artifacts they actually contain. In particular, we focused on virtual software classifications, which are special classifications that describe how to compute their elements. This makes them more abstract, more compact, more expressive and more intentional, than classifications which explicitly enumerate their elements. Also, such an intentional representation is more robust towards change.

In addition to the notion of virtual classifications, virtual dependencies also played an important role in our layered formalism. We can conclude from our case study that virtual dependencies provide a powerful way of defining highly abstract relationships among architectural concepts, by building them up from lower-level relationships that are again constructed from even lower level ones. As such, simple low-level relationships can be successfully combined into complex high-level relationships.

Multiple cross-cutting architectural views

Many traditional approaches towards software architecture assume a more or less direct mapping of the architectural entities to implementation artifacts. During our case study we observed that multiple, potentially overlapping, architectural views with a cross-cutting mapping to the implementation may provide a better insight in the overall structure, organization and functionality of the implementation of a software system. In fact, this observation can be decomposed into two different claims:

1. A software system does not necessarily have one single dominant architecture, but may be described by several, potentially overlapping, architectural views, each providing their own perspective on the implementation.

2. The elements in an architectural view do not necessarily need to correspond directly to implementation artifacts but may cross-cut the implementation structure.

Although it was not the main goal of this dissertation, our case study seems to validate these claims (also see [48]). We defined multiple architectural views on the same software system, each providing their own perspective on the implementation of that system. Not only were these views partially overlapping, in the sense that they described different aspects of the same system, some of the concepts in these views were cross-cutting the implementation, in the sense that they corresponded to implementation artifacts that were distributed across the entire implementation.

Our ability to elegantly express such cross-cutting mappings was a consequence of the followed LMP approach (and of the choice of virtual classifications and virtual dependencies as powerful architectural abstractions), thus providing even more evidence of the expressiveness of the approach.

9.4 Future work

We conclude this dissertation by summarizing some future research topics. Some of them were already mentioned in Chapter 8 and are repeated here.

Fine-tuning and optimizing the formalism and tool. Our current architectural formalism, conformance checking algorithm and prototype tool can be enhanced in many ways. Many of these optimizations, enhancements and extensions were discussed throughout the dissertation. In Section 6.4 we discussed the need for supporting architectural styles, correspondences, deviations and sub-architectures. Section 8.2 discussed some memory and time optimizations. Section 8.3 mentioned some other interesting extensions of the tool such as providing a graphical user interface.

Further validation and scalability. Extra case studies need to be carried out to further validate the feasibility, expressiveness and ease of use of the proposed conformance checking formalism and tool. We are planning to conduct a large case study in an industrial context. Such a case study can also serve as a vehicle to study the scalability of the approach.

Incremental conformance checking. The incremental conformance checking algorithm that was proposed in Section 8.1 should be worked out in more detail, and should be incorporated in the current formalism and tool. Some evolution experiments (both architectural evolution and implementation evolution) need to be carried out to validate the incremental algorithm.

Synchronization and co-evolution. In addition to an incremental conformance checking algorithm, we also need to study support for co-evolution of, and synchronization between, architecture and implementation.

Integration with other tools. In Section 8.3 we mentioned a whole range of tools that could support an architecture-driven development process. Our conformance checking tool should be integrated with all these tools (e.g., the Classification Browser, the Refactoring Browser, a graphical ADL tool and many SOUL tools). A Smalltalk environment is the most obvious choice for this integration effort, as Smalltalk prototypes of most of these tools exist.

Other architectural tools. Whereas prototypes exist for many of the architectural tools mentioned in Section 8.3, this is not the case for all of them. Some of these tools still need to be studied or worked out in more detail. E.g., a tool for reverse-engineering software architectures from the implementation, a partial conformance checking tool, a tool for semi-automatically resolving conformance conflicts, a code generation tool, etc.

Using dynamic information. In this dissertation, a static conformance checking approach was adopted which reasoned about the static software structure only. It should be investigated how the approach can be extended to reason about dynamic information as well.

Generalizing the formalism. As explained in Section 8.4, our conformance checking approach could be generalized to allow architectural conformance checking of software systems written in other object-oriented languages (e.g., Java), other programming languages (e.g., a logic programming language such as Prolog), design languages (e.g., UML), and so on. These generalizations should be implemented, and validated on case studies.

A more difficult generalization is to support architectural conformance checking of hybrid software systems with different parts implemented in different programming languages.

Relation to conceptual graphs. We pointed out before that there are some syntactic similarities between our ADL and the theory of conceptual graphs [75]. This resemblance should be studied in more detail. For example, the theory of conceptual graphs (and in particular, its notion of 'canonical graphs' and 'canonical formation rules') may be a useful candidate to model architectural styles and patterns, and to serve as a formal foundation for compliance checking of architectures to architectural styles [50]. Also, as in conceptual graphs, we could allow concepts and relations to be typed. By mapping relation types to virtual classifications and dependencies, we could enforce different instances of the same relation (e.g., Asks_1 and Asks_2) to have the same denotation. The type hierarchy can also be used for two kinds of evolution — strengthening and weakening —- which correspond to type specialization and generalization, respectively. Finally, 'coreference links' could be used to show corresponding concepts in different views.

Appendix A

Syntax of the SOUL Language

In this appendix, the syntax of the SOUL language is presented in EBNF format.

clause	=	$fact \mid rule \mid query \mid clauses$
fact	=	'**Fact**' $term$ '.'
rule	=	'**Rule**' $regularCompound$ '**if**' $terms$ '.'
query	=	'**Query**' $terms$ '.'
clauses	=	$(clause)^+$
term	=	$simpleTerm \mid compoundTerm \mid specialTerm$
simpleTerm	=	$constantTerm \mid variableTerm \mid booleanTerm$
constantTerm	=	$word$
variableTerm	=	$normalVariable \mid unnamedVariable$
normalVariable	=	'?' $word$
unnamedVariable	=	'_'
booleanTerm	=	'**true**' | '**false**' | '**fail**'
compoundTerm	=	$regularCompound \mid listTerm$
regularCompound	=	$simpleTerm$ '(' $possiblyEmptyTerms$ ')'
listTerm	=	$regularList \mid partialList$
regularList	=	'<' $possiblyEmptyTerms$ '>'
partialList	=	'<' $terms$ '|' $(variableTerm \mid listTerm)$ '>'
specialTerm	=	$smalltalkTerm \mid smalltalkMetaPredicate \mid cutTerm$
smalltalkTerm	=	'[' "extended smalltalk code" ']'
smalltalkMetaPredicate	=	'{' "extended smalltalk code" '}'
cutTerm	=	'!'
possiblyEmptyTerms	=	$terms \mid \epsilon$
terms	=	$(term$',' $)^*$ $term$

Appendix B

Smalltalk Best Practice Patterns

One of the main bottlenecks in software engineering is human communication. Software architectures try to address that problem at a global level by providing a simple mental picture of the overall structure of a software system. Mapping this global structure to the detailed implementation is far from trivial. To discover the intent of a programmer, we often have to wade through piles of documentation and code. However, by using commonly accepted coding conventions and design patterns, it becomes much easier to recognize the intent of a programmer. Therefore, such conventions and patterns can provide important intermediate abstractions in terms of which the architectural mapping can be defined. In this appendix, we discuss some of K. Beck's Smalltalk best practice patterns [5] and discuss how they can be codified in logic predicates as part of our declarative framework (also see [54, 86]), or how they can be of use when defining particular architectural abstractions. (Most definitions of best practice patterns in this Appendix are taken literally from [5].)

Beck's book on Smalltalk best practice patterns can be considered as a kind of style guide, describing the coding conventions that are commonly used by experienced Smalltalk programmers. It addresses topics such as how to choose names for objects, variables and methods, how to clearly communicate certain intents through code, how to split up methods, and so on. 92 patterns are discussed, subdivided in five categories:

Behavior Patterns for methods and messages.

State Patterns for using instance variables and temporary variables.

Collections The major collection classes and messages in the form of patterns.

Classes Patterns for classes.

Formatting Code formatting rules.

We structure this Appendix according to the same set of categories. For each category, we discuss the most relevant patterns. Only formatting patterns are not discussed. Although we agree with Beck that formatting can convey a lot of information on the structure of code, this is mainly so for human readers. Computers have much less problems understanding or analyzing complex structures. They typically ignore all formatting and focus on the structure itself. For example, in our experiments we directly work with parse trees in which no formatting information remains.[1]

[1] This does not imply that formatting cannot provide extra information on a programmer's intentions. We just do not consider this extra information in our experiments.

B.1 Behavior

The first category we discuss are the behavior patterns. Behavior patterns tell programmers how to specify behavior so that their intent is clearly communicated to the reader.

B.1.1 Methods

First, we focus on the method patterns. A programmer should write his or her methods so that they perform the necessary behavior and so that they reveal the intent of the work being done. According to Beck, carefully breaking a computation into methods and carefully choosing their names communicates more about a programmer's intentions than any other programming decision, besides class naming.

Composed Method

The Composed Method pattern states how a Smalltalk program should be divided into methods:

> *Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction. This will naturally result in programs with many small methods, each a few lines long.*

This pattern is typically used in combination with the Intention Revealing Selector pattern so that the different methods are given an easy-understandable name which reveals their intention. The opportunity to communicate through intention revealing method names is the most compelling reason to keep methods small. Small, clearly identified methods are much easier to understand than large ones that do many things at the same time. They allow us to isolate assumptions and intentions, which is essential when we are defining architectural mappings. For example, as will be illustrated in Subsection B.1.2 when we discuss the Intention Revealing Selector pattern, it allows us to define useful virtual classifications by grouping all methods with a similar name.

Constructor Method

The Constructor Method pattern suggests how to represent instance-creation methods:

> *Provide methods that create well-formed instances. Pass all required parameters to them. Put these methods in a method protocol called '`instance creation`'.*

The fact that all instance-creation methods are, by convention, put in the '`instance creation`' method protocol, makes it very easy to define a predicate `instanceCreationMethod` which recognizes constructor methods (see below) or to define a predicate `isCreatedBy_C_C` which checks for an instance-creation relationship (see Subsection 7.1.5).

```
% Is Method an instance-creation method for Class?
instanceCreationMethod(Class, Method) :-
    metaClass(Class, Meta),
    creationProtocolName(ProtocolName),
    protocolName(Meta, Protocol, ProtocolName),
    methodInProtocol(Meta, Protocol, Method),
    returnType(Method, Class).
```

The auxiliary predicate `creationProtocolName` was defined in Subsection 7.1.5 on page 131. All other auxiliary predicates were discussed in Subsection 5.3.5.

Method naming and tagging conventions

The previous pattern is interesting because it specifies a tagging convention which easily allows us to recognize constructor methods. Beck mentions many other method patterns that specify simple naming and tagging conventions which allow us to recognize certain kinds of methods. The commonalities in all these patterns are captured by the following generic logic predicate:

```
recognizeMethod(Method, StringPattern, ProtocolName) :-
    methodName(Method, MethodName),
    patternMatch(MethodName, StringPattern),
    protocolName(Protocol, ProtocolName),
    methodInProtocol(_, Protocol, Method).
```

It checks whether some `Method` has a name that matches a certain `StringPattern` and whether the `Method` belongs to a method protocol named `ProtocolName`.

Some of these method patterns that specify simple naming and tagging conventions are listed below. For each method pattern in that list, we specify how the `recognizeMethod` predicate needs to be instantiated to check the format for that particular pattern.

- **Constructor Method.** As mentioned above, by convention, every constructor method is put in the '`instance creation`' method protocol. Its name can be anything.

  ```
  constructorMethodFormat(Method) :-
      recognizeMethod(Method, anything, 'instance creation').
      % 'anything' is a wildcard pattern
  ```

- **Converter Method.** How does one represent simple conversion of an object to another with the same protocol but different format? *Provide a method in the object to be converted that converts to the new object. Name the method by prefixing '*`as`*' to the class name of the object returned. Put the method in a method protocol called '*`private`*'.*

  ```
  converterMethod(Method, Type) :-
      returnType(Method, Type),
      converterMethodFormat(Method, Type).

  converterMethodFormat(Method, Type) :-
      recognizeMethod(Method, pattern(['as',TypeName]), 'private'),
      className(Type, TypeName).
  ```

- **Query Method.** How do you represent testing a property of an object? *Provide a method that returns a Boolean. Name it by prefacing the property name with '*`is`*'. Put the method in a protocol called '*`testing`*'.* If you use the logical inverse of a Query Method a lot, also provide an inverse method. Name this inverse method by prefacing it with '`not`', or try to find a positive way of saying the inverse (in which case the '`is`' prefix should be used).

  ```
  queryMethod(Method) :-
      returnType(Method, Type),
      className(Type, 'Boolean'),
      queryMethodFormat(Method).

  queryMethodFormat(Method) :-
      recognizeMethod(Method, or(prefix('is'),prefix('not')), 'testing').
  ```

- **Debug Printing Method.** How do you code the default printing method? *Override `printOn:` to provide information about an object's structure to the programmer. Put printing methods in the method protocol '`printing`'.*

```
debugPrintingMethod(Method) :-
    overridenMethod(Method),
    debugPrintingMethodFormat(Method).

debugPrintingMethodFormat(Method) :-
    recognizeMethod(Method, exact('printOn:'), 'printing').
```

- **Converter Constructor Method.** How does one represent the conversion of an object to another with different protocol? *Make a Constructor Method that takes the object to be converted as an argument. Name the method by prefixing '`from`' to the class of the object being converted. Put the method in a method protocol called '`instance creation`'.*

```
converterConstructorMethod(Method, Class) :-
    hasParameterType_M_C(Method, Class),
    converterConstructorMethodFormat(Method, Class).

converterConstructorMethodFormat(Method, Class) :-
    recognizeMethod(Method, pattern(['from',ClassName]), 'instance creation'),
    className(Class, ClassName).
```

- **Constructor Parameter Method.** How do you set instance variables from the parameters to a Constructor Method? *Code a single method that sets all the variables. Preface its name with '`set`', then the names of the variables. Put this method in a method protocol called '`private`'.*

```
constructorParameterMethod(Method, Class) :-
    classImplementsMethod(Class, Method),
    constructorParameterMethodFormat(Method, Class).

constructorParameterMethodFormat(Method, Class) :-
    recognizeMethod(Method, pattern(['set'|InstVarNames]), 'private'),
    instVarNames(Class, InstVars),   % retrieve list of inst. var. names
    addColons(InstVars, Names),      % add a colon after every inst. var. name
    permutation(Names, InstVarNames).
```

Return types

Various types of method patterns also suggest return types. For example:

- **Query Method** suggests that the return type is a Boolean.
- **Constructor Method** returns objects of the class on which the constructor method is defined. (Same for Converter Constructor Method which is a special kind of Constructor Method.)
- **Converter Method** states that the method name is '`as`' appended with the class of the object returned.

This information can be used to optimize the predicate `returnType` which infers the return type of some method. If the method has one of the above formats we can readily extract its type, in all other cases we use the original non-optimized version of the predicate.

```
returnTypeOptimized(Method,Type) :-
  queryMethodFormat(Method)           -> className(Type, 'Boolean');
  constructorMethodFormat(Method)     -> classImplementsMethod(Type, Method);
  converterMethodFormat(Method, Type) -> true;
  otherwise                           -> returnType(Method,Type).
```

Method comments

About method comments, Beck writes that they are not often used in Smalltalk. This is mainly because there exist commonly accepted coding conventions that let a developer communicate tactical information without any supporting comments. Most information that could be provided by a method comment is already captured in the code with various patterns. As (has been or will be) discussed, Intention Revealing Selector communicates what the method does; Type Suggesting Parameter Name says what the arguments are expected to be; and various types of method patterns suggest return types, like Query Method for methods returning Booleans.

Type Suggesting Parameter Name

Beck categorized this pattern as a formatting pattern. We prefer to categorize it as a behavior pattern, because it is about how to name the parameter of a method. Two pieces of information are important for every variable: its type and the role it plays in the computation. Method keywords communicate the role of some method parameter. Argument types are suggested by providing an appropriate parameter name:

> *Name parameters according to their most general expected class, preceded by 'a' or 'an'. If there is more than one parameter with the same expected class, precede the class name with a descriptive word.*

In Subsection 5.3.5, we mentioned some rather computation-intensive predicates to infer the potential type of certain Smalltalk expressions. The above naming convention may provide a low-cost alternative for guessing the type of method parameters. (Or we can use a combination of both approaches to infer the most likely type among a set of candidate classes.) We merely need to take the postfix of a parameter name (e.g., using the predicate `stringEndsWith`) to know its type.

B.1.2 Messages

Beck's message patterns describe some tactical ways in which messages can be used. They provide a set of common techniques for solving problems by manipulating the communication between objects. We discuss some of these patterns here.

Intention Revealing Selector — Intention Revealing Message

We already mentioned the Intention Revealing Selector pattern while discussing the Composed Method pattern in Subsection B.1.1. It tells us how to name a method:

> *Name methods after what they accomplish.*

Rather than naming a method after *how* it accomplishes a task, it should be named after *what* it is supposed to accomplish. The 'how' can always be derived from the method body itself. Naming methods like this reveals a lot of the programmer's intentions. The closely related Intention Revealing Message pattern essentially states the same as the Intention Revealing Selector pattern, but from the point of view of method invocation (as opposed to the method definition). It tells a developer how to communicate his intent when sending a message:

> *Send a message to '`self`'. Name the message so that it communicates what is to be done rather than how it is to be done. Code a simple method for the message.*

By choosing Intention Revealing method names, it may become very easy to find all methods that correspond to a certain architectural concept. For example, the **Query Interpreter** architectural concept in the 'user interaction' view conceptually represents the interpretation process. It is mapped to the set of all methods that implement this interpretation process. All

these methods have names like `interpret:repository:`, representing what they are supposed to accomplish, thus making it easy to identify the relevant methods. Therefore, we could have defined the mapping by merely grouping all methods that have a name with some form of the verb 'interpret' in it. However, because all these methods were also tagged with the same method protocol 'interpretation' or 'interpreting', which also clearly expresses their intent, we decided to define the grouping based on that information instead.

Double Dispatch

In Smalltalk, when a message with some arguments is sent to an object, only the class of the receiver is taken into account when looking for a corresponding method. In some cases, though, we want the behavior to be invoked to depend not only on the class of the receiver, but on the class of one of the arguments as well. The Double Dispatch coding pattern provides a clean solution to this problem:

> *Send a message to the argument. Append the class name of the receiver to the selector. Pass the receiver as an argument.*

The predicate below codifies the Double Dispatch coding pattern.[2]

```
doubleDispatchMethod(Method) :-
    classImplementsMethodNamed(Class, MN, Method),
    className(Class, CN),
    methodArgument(Method, Argument),
    argumentVarName(Argument, VarName),
    findMethod(Class, Method,
               pattern(['return(send(',VarName,',',MN,CN,',[variable(self)]))']))).
```

A similar predicate could be defined to verify whether two methods communicate with each other according to a double dispatch protocol.

Other communication protocols

In addition to the Double Dispatch communication protocol, Beck discusses many other communication protocols among methods, such as:

- **Extending Super.** How do you add to a superclass' implementation of a method? *Override the method and send a message to '`super`' in the overriding method.*
- **Simple Delegation.** How do you invoke a disinterested delegate? *Delegate messages unchanged.*
- **Self Delegation.** How do you implement delegation to an object that needs reference to the delegating object? *Pass along the delegating object (i.e., '`self`') in an additional parameter called '`for:`'.*
- **Pluggable Selector.** How do you code simple instance-specific behavior? *Add an instance variable that contains a selector to be performed. Append '`Message`' to the Role Suggesting Instance Variable Name. Create a Composed Method that simply performs the selector.*
- **Pluggable Block.** How do you code complex pluggable behavior that is not quite worth its own class? *Add an instance variable to store a block. Append '`Block`' to the Role Suggesting Instance Variable Name. Create a Composed Method to evaluate the block to invoke the pluggable behavior.*

[2] The predicate actually codifies only a very specific case of the Double Dispatch pattern, where the double dispatch method has exactly one argument. Though a bit more complex, the pattern is similar for methods with multiple arguments.

- **Collecting Parameter.** How do you return a collection that is the collaborative result of several methods? *Add a parameter that collects their results to all of the submethods.*

All these different protocols can be codified in logic predicates. We will not show the implementation of all these predicates. As an illustration, we only present the implementation of a predicate `simpleDelegationMethod` which codifies the Simple Delegation communication pattern.

```
simpleDelegationMethod(Method) :-
    methodName(Method, Message),
    methodArgumentsString(Method, Arguments),
    findMethod(_, Method,
               pattern(['send(',_Delegate,',',Message,',',Arguments,')'])).
```

where the auxiliary predicate `methodArgumentsString` produces a string representing the argument list of some method:

```
methodArgumentsString(Method, ArgumentsString) :-
    % compute list of argument names for the method
    findall( VarName,
             ( methodArgument(Method, Var), argumentVarName(Var, VarName) ),
             ArgumentList ),
    % convert this list to a string
    list_string(ArgumentList,ArgumentsString).
```

B.2 State

In Smalltalk, as well as in other object-oriented languages, behavior is considered more important than state. However, the tactical decisions a programmer makes about representing state also have an important impact on the quality and readability of his or her code. Following Beck [5], this section considers two kinds of state: instance variables and temporary variables. We start with the former.

B.2.1 Instance variables

Common State — Role Suggesting Instance Variable Name

How do you represent state, different values for which will exist in all instances of a class?

> *Declare an instance variable in the class.*

Instance variables have a very important communicative role to play. A set of objects reveals a lot that was in the mind of the original programmer just by what the instance variables are and what they are named. As was the case for methods, when the instance variable names are well chosen, this allows us to define useful virtual classifications by grouping all variables with a similar name. The Role Suggesting Instance Variable Name pattern suggests how to name an instance variable:

> *Name instance variables for the role they play in the computation. Make the name plural if the variable will hold a Collection.*

Explicit Initialization — Lazy Initialization

To initialize instance variables to their default value, there are two alternatives: Explicit Initialization or Lazy Initialization. Both have their advantages and disadvantages. For a closer comparison of both alternatives we refer to [5].

With Explicit Initialization the values of instance variables are initialized explicitly by some initialization method:

> *Implement a method '`initialize`' that sets all the values explicitly. Override the class message '`new`' to invoke it on new instances. Put '`initialize`' methods in a method protocol called '`initialize-release`'.*

The following predicate checks whether some `Method` is an Explicit Initialization Method.

```
explicitInitializationMethod(Method) :-
    explicitInitializationMethodFormat(Method),
    classImplementsMethod(Meta, Method),
    metaClass(Class, Meta),
    forall( ( instVar(Class, InstVar), instVarName(InstVar, VarName) ),
            findMethod(Meta, Method, pattern([_,'assign(variable(',VarName,'),',_]))
            ).

explicitInitializationMethodFormat(Method) :-
    recognizeMethod(Method, exact('initialize'), 'initialize-release').
```

With Lazy Initialization, initialization is done lazily through some accessor method. The variable is given some default value the first time the accessor is invoked. After that, the current value of the variable is simply returned. The fact that the variable has not yet been initialized can be recognized because it still contains a nil value.

> *Write an accessor method for the variable, which initializes the variable if necessary with some default value.*

The predicate `lazyInitialisedAccessorMethod(Method, InstVar)` below verifies whether `Method` is a Lazy Initialization Accessor Method for some instance variable `InstVar`:

```
lazyInitialisedAccessorMethod(Method, InstVar) :-
    classImplementsMethodNamed(Class, MethodName, Method),
    className(Class, ClassName),
    instVar(Class, InstVar),
    instVarName(InstVar, IVName),
    methodParseTree(ClassName, MethodName, [], _,
                    [return(send(NilCheck,'ifTrue:ifFalse:',[_TrueBlock,FalseBlock]))]),
    nilCheckStatement(NilCheck, variable(IVName)),
    blockStatements(FalseBlock,[variable(IVName)]).

% nilCheckStatement defines the possible forms of a nil check statement
nilCheckStatement(send(Var,'isNil',[]), Var).
nilCheckStatement(send(Var,'==',[literal('nil')]), Var).
nilCheckStatement(send(Var,'=',[literal('nil')]), Var).

% blockStatements extracts the statementlist from a block
blockStatements(block(arguments(_),temporaries(_),statements(Statements)), Statements).
```

Direct/Indirect Variable Access — Getting and Setting Method

A first way to get and set the values of instance variables is to use the variables directly in all the methods that need their values. This is what we call Direct Variable Access:

> *Access and set the variable directly.*

The alternative to Direct Variable Access is Indirect Variable Access. Instead of directly accessing the instance variable, a message is sent every time the variable needs to be used or changed:

> *Access and set the value of instance variables only through a Getting or Setting Method.*

Getting and Setting Methods are also known as *accessing* methods. A Getting Method, or *accessor* method, specifies how to provide read-access to an instance variable:

> *Provide a method that returns the value of the variable. Give it the same name as the variable. Put private Getting Methods in a method protocol called '`private-accessing`'. Put public Getting Methods in a method protocol called '`accessing`'.*

As before, a convention like this one allows us to quickly recognize accessor methods based on their naming and tagging convention.

```
accessorMethod(Method) :-
    classImplementsMethod(Class, Method),
    accessor(Class, Method, _VarName).

accessor(Class, Method, VarName).
    instVar(Class, InstVar),
    instVarName(InstVar, VarName),
    accessorMethodFormat(Method, VarName),
    findMethod(Class, Method, pattern(['return(variable(',VarName,'))'])).

accessorMethodFormat(Method, VarName) :-
    accessingProtocol(ProtocolName),
    recognizeMethod(Method, exact(VarName), ProtocolName).

accessingProtocol('accessing').
accessingProtocol('private-accessing').
```

A Setting Method, or *mutator* method, specifies how to update the value of an instance variable:

> *Provide a method that assigns a value to the variable. Give it the same name as the variable, appended with a colon. Put private Setting Methods in a method protocol called '*`private-accessing`*'. Put public Setting Methods in a method protocol called '*`accessing`*'.*

The Prolog code which codifies this pattern is presented below:

```
mutatorMethod(Method) :-
    classImplementsMethod(Class, Method),
    mutator(Class, Method, _VarName).

mutator(Class, Method, VarName) :-
    instVar(Class, Variable),
    instVarName(Variable, VarName),
    mutatorMethodFormat(Method, VarName),
    findMethod(Class, Method, pattern(['assign(variable(',VarName,'),',_,')']) ).

mutatorMethodFormat(Method, VarName) :-
    accessingProtocol(ProtocolName),
    recognizeMethod(Method, pattern([VarName,':']), ProtocolName).
```

Finally, by combining the `accessorMethod` and `mutatorMethod` predicates, we can define a predicate which checks for an accessing method (i.e., a Getting or Setting method):

```
accessingMethod(Method) :-
    accessorMethod(Method);
    mutatorMethod(Method).
```

B.2.2 Temporary variables

The patterns that deal with temporary variables are about how to store and reuse the value of expressions in a method body, about how to improve the performance or readability of methods, etc. They are very local and low-level and therefore of little interest for architectural purposes. Therefore, we do not discuss any of these patterns.

B.3 Collections

One of the great strengths of Smalltalk is that it offers a unified protocol to all the varieties of ways of representing one-to-many relationships. The **Collection** pattern states:

> *To represent a one-to-many relationship, use a collection.*

The following best practice patterns summarize the uniform collection protocol:

- **Enumeration.** *Use the enumeration messages to spread a computation across a collection.*
- **Do.** *Send* `do:` *to a collection to iterate over its elements. Send a one-argument block as the argument to* `do:`*. It will be evaluated once for each element.* For purposes of enumeration, there is no difference between the collection classes in Smalltalk. You just send the message '`do:`'.
- **Collect.** How do you operate on the result of a message sent to each object in a collection? *Use* `collect:` *to create a new collection whose elements are the results of evaluating the block passed to* `collect:` *with each element of the original collection. Use the new collection.*
- **Select/Reject.** How do you filter out part of a collection? *Use* `select:` *and* `reject:` *to return new collections containing only elements of interest. Both take a one-argument block that returns a Boolean.* `select:` *gives you elements for which the block returns true,* `reject:` *gives you elements for which the block returns false.*
- **Detect.** *Search a collection by sending it* `detect:`*. The first element for which the block argument evaluates to true will be returned.* There is a variation of `detect:`, `detect:ifNone:`, that takes an additional zero-parameter block as an argument. This variation is useful if you are not sure any element will be found.
- **Inject:into:** *Use* `inject:into:` *to keep a running value as you iterate over a collection. Make the first argument the initial value. Make the second argument a two element block. Call the block arguments 'sum' and 'each'. Have the block evaluated to the next value of the running value.*

By using such a common set of messages to manipulate collections of elements, client code is effectively decoupled from decisions about how to store a collection of elements. The following set of facts defines the messages that are typically used for enumerating over such collections:

```
enumeratorMessage('do:').
enumeratorMessage('collect:').
enumeratorMessage('select:').
enumeratorMessage('reject:').
enumeratorMessage('detect:').
enumeratorMessage('detect:ifNone:').
enumeratorMessage('inject:into:').
```

Based on this knowledge, we can easily define a logic rule that codifies the typical structure of a one-to-many statement. This is useful, for example, when declaratively codifying the Composite design pattern. In [86], Wuyts defines a rule `oneToManyStatement(Method,InstVar)` which states that a method `Method` contains a one-to-many relation if it enumerates over a collection held in an instance variable `InstVar`.

B.4 Classes

There is probably no coding decision with more effect on the quality of the code than the names that are given to classes. Good class names provide insight into the purpose and design of a system. Beck proposes the following class naming conventions:

- **Simple Superclass Name.** *Name a class that is expected to be the root of an inheritance hierarchy with a single word that conveys its purpose in the design.* For example: `Number`, `Collection`, `Magnitude`, `Model`.
- **Qualified Subclass Name.** *Name subclasses in an inheritance hierarchy by prepending an adjective to the superclass name.* For example: `OrderedCollection`, `SortedCollection`, `LargeInteger`. (Note that, if inheritance is used strictly for code sharing and the role of the subclass is different from the role of the superclass, we still use the Simple Superclass Name convention.)

When naming conventions such as these are used, it is much more easy to understand the code and to define its mapping to the architecture. The name of a class often provides an indication of the architectural concept(s) it may correspond to. For example, in the implementation of the SOUL system, all classes representing repositories end with the string '`Repository`'. Based on this convention, we could define the **Repository** architectural concept by means of a virtual classification which computes all classes with such a name. (Although we adopted an alternative mapping scheme which declares that all repository classes inherit from the same abstract superclass '`SOULAbstractRepository`'.)

B.5 Summary

To summarize this Appendix, Table B.1 shows a list of predicates codifying some of the Smalltalk best practice patterns that were discussed in this chapter. In fact, all these predicates belong to the coding conventions layer of the DFW, and should therefore be merged with Table 5.6 on page 71 (Subsection 5.3.5).

Predicate name and arguments	**Meaning of the predicate**
Behavior category — Method patterns	
recognizeMethod(M,Pa,Pr)	*generic predicate to pattern match a method's name and check its protocol*
instanceCreationMethod(C,M)	M is Instance Creation Method for class C
converterMethod(M,C)	M is Converter Method to class C
queryMethod(M)	M is Query Method
debugPrintingMethod(M)	M is Debug Printing Method
converterConstructorMethod(M,C)	M is Converter Constructor Method from C
constructorParameterMethod(M,C)	M is Constructor Parameter Method for C
returnTypeOptimized(M,C)	M returns object of class C
Behavior category — Message patterns	
doubleDispatchMethod(M)	method M uses Double Dispatch
simpleDelegationMethod(M)	method M uses Simple Delegation
State category — Instance variable patterns	
explicitInitializationMethod(M)	M is Explicit Initialization Method
lazyInitialisedAccessorMethod(M,V)	M is Lazy Initialization accessor method for some instance variable V
accessor(C,M,V)	method M of class C gets value of variable V
accessorMethod(M)	M is Getting Method
mutator(C,M,V)	method M of class C updates value of variable V
mutatorMethod(M)	M is Setting Method
accessingMethod(M)	M is Getting or Setting Method
Collection category patterns	
enumeratorMessage(N)	message N is an Enumeration message
oneToManyStatement(M,V)	method M implements a one-to-many relationship

Table B.1: Some predicates codifying Smalltalk best practice patterns.

Appendix C

Terminology

ADL See architecture description language.

AML See architectural mapping language.

Architectural formalism In this dissertation, when we talk about the architectural formalism, we mean the formalism that is explained in Chapter 5. That is, the architectural language in which to describe the conceptual architecture and its mapping to the implementation, as well as the conformance checking algorithm that is defined in terms of the constructs provided by this architectural language.

Architectural abstraction In our conformance checking approach, architectural abstractions are the intermediary abstractions that define the actual mapping of architectural entities to implementation artifacts and their dependencies.

Architectural abstraction language The architectural abstraction language, which is part of the AML, provides intuitive high-level abstractions of sets of implementation artifacts and their dependencies that can straightforwardly be mapped to the different kinds of architectural entities of the ADL.

Architectural concept Instead of talking about architectural 'components', in this dissertation we use the term 'architectural concept'. This corresponds to our intuition that a software architecture expresses relations (or structure) over abstract concepts that have some meaning for the application domain.

Architectural conformance checking The task of verifying whether the implementation structure of some software system corresponds to the more abstract structure described by its conceptual architecture.

Architecture description An architecture description is an explicit description of the structure of some conceptual architecture. Architecture descriptions are described in an ADL.

Architecture description language (ADL) An ADL provides a formal notation in which software architectures can be described explicitly, by specifying the syntax and semantics of the architectural entities and their interactions.

The ADL used in this dissertation essentially describes the structure of the architecture (i.e., its syntax). The semantics of the different architectural entities will be described implicitly in terms of how they are mapped to the implementation. We do this in a separate language, the AML.

Architectural instantiation An architectural instantiation associates architectural entities with intermediary abstractions defined in the architectural abstraction language.

Architectural instantiation language In the architectural instantiation language, which is part of the AML, we can map architectural entities defined in the ADL to intermediary abstractions defined in the architectural abstraction language.

Architecture language The architecture language describes what a conceptual architecture looks like and describes how the different architectural entities are mapped to the implementation. For this purpose, the architecture language is split into an ADL and an AML.

Architectural mapping Architectural mappings are declared in the AML. An architectural mapping consists of two parts: an architectural instantiation and an architectural abstraction.

Architectural mapping language (AML) The AML allows us to codify the mapping to the implementation for each of the architectural views described in the ADL, thus defining the meaning of the different architectural entities in each of these views. Every architectural entity is defined in terms of implementation artifacts and their dependencies.

Architectural relation Instead of talking about architectural 'connectors', in this dissertation we use the term 'architectural relation'. Architectural relations describe the relationships among architectural concepts.

Architectural view An architectural view describes the structure of a software system from some conceptual point of view. It consists of a set of architectural concepts and architectural relations together with the links that glue them together.

Conceptual architecture A conceptual architecture describes a software system from multiple high-level architectural points of view, abstracting away from the implementation details of the system. Each architectural view focuses on a different aspect of the structure of the software system.

Conformance conflict When the implementation is not in conformance with its conceptual architecture (i.e., with one of its architectural views), we call this situation an architectural conformance conflict.

Declarative framework (DFW) To define architectural abstractions in a LMP language, an architect can make use of a layered library of predefined logic predicates. We call this library the declarative framework.

DFW See declarative framework.

Filter Filters are the architectural abstractions in terms of which concept ports are defined. A filter selects some subset of a software classification.

Implementation artifact In our LMP approach, when we use the term 'implementation artifact', we mean the primitive base-level language constructs that can be manipulated and reasoned about at meta level in the logic programming language.

For example, when using Smalltalk as a base-language, the implementation artifacts are classes, meta classes, instance variables, class variables, method arguments, temporary variables, etc.

Incremental conformance checking With an incremental conformance checking approach, instead of having to re-check conformance for the entire implementation and architecture (when either the implementation or the architecture has evolved), we only need to analyze those parts that were affected by the evolution.

Link In an architectural view, architectural concepts are connected to architectural relations by linking ports to roles.

LMP See logic meta programming.

Logic meta programming (LMP) LMP is the use of a logic programming language at meta level to reason about implementation artifacts and their dependencies in some base language.

In this dissertation, we use a Prolog-like logic language (i.e., SOUL or Prolog) at meta level and an object-oriented language (i.e., Smalltalk) at base level.

Port Ports represent the interface of architectural concepts. Any concept may have multiple ports.

Quantifier In the context of the architectural formalism proposed in this dissertation, a quantifier specifies how to apply some logic relation over a set of elements. (It 'quantifies' the relation over the elements in the set.)

Role Roles represent the interface of an architectural relation. They identify the required participants for that relation.

Software architecture A software architecture is commonly defined as a collection of components, together with a description of interactions and relationships among those components (the architectural connectors), and optionally a set of constraints on these components and connectors.

In this dissertation, we call the components 'architectural concepts', and the connectors 'architectural relations'. Furthermore, instead of using the term software architecture we make a distinction between a 'conceptual architecture' and an 'architectural view'. This is because we allow a software system to be described by multiple, potentially overlapping architectural views. Architectural views are software architectures in the sense that they are described in terms of concepts and relations. A conceptual architecture is the union of all architectural views and provides a more complete picture of the software architecture of some software system.

Software classification A software classification is a set of related implementation artifacts. Artifacts can be classified in multiple classifications.

SOUL The Smalltalk Open Unification Language, SOUL, is a hybrid logic programming language, implemented in Smalltalk and with a tight symbiosis with both the Smalltalk language and development environment. The syntax of the language is similar to that of the logic programming language Prolog, but has an extension that allows meta-level reasoning about Smalltalk code.

Virtual dependency In our LMP approach, virtual dependencies correspond to logic predicates that describe high-level implementation or design relationships among implementation artifacts.

Virtual classification A virtual (software) classification is a software classification that is specified intentionally (i.e., in terms of a declarative description from which its elements can be computed), as opposed to extensionally (i.e., by explicitly enumerating its elements).

In our LMP approach, we represent virtual classifications as logic predicates that compute a set of implementation artifacts.

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill Book Company, 1985.

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-based Distributed Processing*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer-Verlag, 1993.

[3] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison Wesley Longman, 1998.

[5] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[6] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[7] I. Borne, S. Demeyer, and G. H. Galal. Object-oriented architectural evolution. In A. Moreira and S. Demeyer, editors, *ECOOP 1999 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, 1999.

[8] M. Broy. A uniform mathematical concept of a component. *Software — Concepts & Tools*, 19(1):57–59, 1998.

[9] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plášil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What characterizes a (software) component? *Software — Concepts & Tools*, 19(1):49–56, 1998.

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

[11] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th international workshop on software specification and design*, pages 16–25. IEEE Computer Society Press, March 1996.

[12] K. De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1998.

[13] K. De Hondt and P. Steyaert. Exploiting classification for software evolution. Position paper, MediaGeniX, March 2000. ECOOP 2000 Workshop on Objects and Classification: a Natural Convergence.

[14] K. De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1998.

[15] K. De Volder. Aspect-oriented logic meta programming. In *Proceedings of International Reflection 1999 Conference*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer-Verlag, 1999.

[16] D. Deridder and B. Wouters. The use of ontologies as a backbone for software engineering tools. In *Proceedings of the Fourth Australian Knowledge Acquisition Workshop AKAW99*, pages 187–200, 1999. December 5-6, Sydney, Australia.

[17] M. D'Hondt, W. De Meuter, and R. Wuyts. Using reflective programming to describe domain knowledge as an aspect. In *Proceedings of GCSE 1999*, 1999.

[18] M. D'Hondt and T. D'Hondt. Is domain knowledge an aspect? In *Proceedings of the ECOOP 1999 Aspect-Oriented Programming Workshop*, 1999.

[19] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of SACT 2000*. Kluwer Academic Publishers, January 2000. International symposium on Software Architectures and Component Technology.

[20] M. Dorfman and R. H. Thayer. *Software Engineering*. IEEE Computer Society Press, 1997.

[21] R. Fairley. *Software Engineering Concepts*. McGraw-Hill, 1985.

[22] S. Fraser, A. Cockburn, L. Brajkovich, J. Coplien, L. Constantine, and D. West. OO anthropology: Crossing the chasm (panel 3). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31(10) of *ACM SIGPLAN Notices*, pages 286–291, New York, October 1996. SIGPLAN, ACM Press. Proceedings of OOPSLA '96.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addisson-Wesley, 1994.

[24] P. K. Garg and W. Scacchi. A hypertext system to manage software life-cycle documents. *IEEE Software*, 7(3):90–98, May 1990. Reprinted in [6].

[25] D. Garlan. First international workshop on architectures of software systems — workshop summary. *ACM SIGSOFT, Software Engineering Notes*, 20(3):84–89, 1995.

[26] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, November 1995.

[27] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON 1997*, pages 169–183, Toronto, Ontario, November 1997.

[28] D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21, April 1995.

[29] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. River Edge, NJ: World Scientific Publishing Company, 1993.

[30] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA 1993*, pages 411–428. ACM Press, 1993.

[31] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, September 1985.

[32] W. L. Hürsch and C. V. Lopes. Separation of concerns. Technical report, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[33] P. Inverardi, A. L. Wolf, and D. Yankelevich. Checking assumptions in component dynamics at the architectural level. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 46–63. Springer-Verlag, September 1997. Second International Conference, COORDINATION 1997, Berlin, Germany.

[34] C. B. Jaktman. A maintenance check for evolving a product-line architecture by determining the indicators of erosion, 1998. Workshop on Empirical Studies of Software Maintenance (WESS98), Bethesda, Maryland, November 16.

[35] C. B. Jaktman, J. Leaney, and M. Liu. Structural analysis of the software architecture — a maintenance assessment case study. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*. Kluwer Academic, 1999. 22-24 February 1999, San Antonio, Texas, USA.

[36] G. Kiczales. Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP 1997*. Springer, 1997. Invited presentation.

[37] J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 18–31. Springer-Verlag, September 1997. Second International Conference, COORDINATION 1997, Berlin, Germany.

[38] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, November 1995.

[39] K. J. Lieberherr. *Adaptive Object-Oriented Software. The Demeter Method with propagation patterns*. PWS Publishing Company, 1996.

[40] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 1997.

[41] C. Lucas, P. Steyaert, and K. Mens. Managing software evolution through reuse contracts. In *Proceedings of the First Euromicro Conference on Software Maintenance and reengineering; Berlin, Germany*. IEEE Computer Society Press, 1997.

[42] R. Lucas. *LPA WIN-PROLOG 4.0 Prodata Interface*. Keylink Computers Ltd., 1997.

[43] D. C. Luckham and J. V. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pages 717–734, September 1995.

[44] B. J. MacLennan. *Principles of Programming Language*. Saunders College Publishing, second edition edition, 1987.

[45] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of ESEC/FSE 1997*, volume 22(6) of *ACM SIGSOFT Software Engineering Notes*. ACM Press, November 1997.

[46] N. Medvidovic, R. N. Taylor, and D. S. Rosenblum. An architecture-based approach to software evolution, 1998.

[47] R. Melton and D. Garlan. Architectural unification. Technical report, School of Computer Science, Carnegie Mellon University, January 1997.

[48] K. Mens. Multiple cross-cutting architectural views. Position paper, Programming Technology Lab, Vrije Universiteit Brussel, February 2000. Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000).

[49] K. Mens and T. Mens. Codifying high-level software abstractions as virtual classifications. Position paper, Programming Technology Lab, Vrije Universiteit Brussel, March 2000. ECOOP 2000 Workshop on Objects and Classification: a Natural Convergence.

[50] K. Mens and M. Wermelinger. On the use of knowledge representation techniques for modeling software architectures. Submitted to the 4^{th} International Software Architecture Workshop (ISAW-4), 4 and 5 June 2000, Limerick Ireland, in conjunction with ICSE 2000, February 2000.

[51] K. Mens, B. Wouters, C. Lucas, A. Grijseels, R. Harmegnies, P. Ravijts, and F. Sylvestre. Compliance checking in object-oriented systems - bi-annual report. Research project report, Programming Technology Lab, Vrije Universiteit Brussel and Wang Global Belgium, September 1999. Progress report submitted to the Brussel's Capital Region.

[52] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS Europe 1999*, pages 33–45. IEEE Computer Society Press, 1999. TOOLS 29 — Technology of Object-Oriented Languages and Systems, Nancy, France, June 7-10.

[53] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 1999.

[54] I. Michiels. Using logic meta programming for building sophisticated development tools. Computer science graduation report, Vrije Universiteit Brussel, Belgium, 1997.

[55] N. H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1), 1996.

[56] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, pages 356–372, April 1995.

[57] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT 1995, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[58] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[59] H. R. Nielson and F. Nielson. *Semantics with Applications — A Formal Introduction*. Wiley Professional Computing, 1993.

[60] OMG ad/99-06-08. *UML Notation Guide version 1.3*, 1999.

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

[62] P. Oreizy. Issues in modeling and analyzing dynamic software architectures. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, 1998. Marsala, Sicily, Italy, June 30.

[63] H. Ossher and P. Tarr. Concern spaces: Structuring systems with hypermodules. Technical report, IBM Thomas J. Whatson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, 1999.

[64] M-C. Pellegrini. Dynamic reconfiguration of CORBA-based applications. In *Proceedings of TOOLS Europe 1999*, pages 329–340. IEEE Computer Society Press, 1999. TOOLS 29 — Technology of Object-Oriented Languages and Systems, Nancy, France, June 7-10.

[65] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[66] J. S. Poulin. Evolution of a software architecture for management information systems. In *Proceedings of the Second International Software Architecture Workshop (ISAW2)*, pages 134–137, 1996. San Francisco, California, USA, 14-15 October.

[67] M. J. Presso. Generic component architecture using meta-level protocol descriptions. Master's thesis, Vrije Universiteit Brussel, 1999. European Masters in Object Oriented Software Engineering.

[68] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, November 1987.

[69] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object systems*, 3(4), 1997.

[70] D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST 1996, Chicago, IL*, April 1996.

[71] N. Romero. Managing evolution of software architectures with reuse contracts. Master's thesis, Vrije Universiteit Brussel, 1999. European Masters in Object Oriented Software Engineering.

[72] R. W. Schwanke, V. A. Strack, and T. Werthmann-Auzinger. Industrial software architecture with Gestalt. In *Proceedings of IWSSD-8*, pages 176–180. IEEE Computer Society Press, 1996.

[73] M. Shaw. Software architecture: A roadmap. Presentation at ICSE 2000, Limerick, Ireland, 2000.

[74] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[75] J. F. Sowa. *Conceptual Structures — Information processing in mind and machine.* The Systems Programming Series. Addison-Wesley, 1984.

[76] L. Steels. *Kennissystemen.* Addison-Wesley Nederland, 1992.

[77] P. Stevens and R. Pooley. *Using UML — Software Engineering with Objects and Components.* Addison Wesley, 1999. Updated edition.

[78] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the OOPSLA 1996 Conference on Object-Oriented Programming, Systems, Languages and Applications*, number 31(10) in ACM SIGPLAN Notices, pages 268–285. ACM Press, 1996.

[79] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press / Addisson-Wesley, 1998.

[80] P. Tarr, H. Ossher, W. Harrison, and Jr. S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE 1999)*, 1999.

[81] T. Tourwé and W. De Meuter. Optimizing object-oriented languages through architectural transformations. In 8^{th} *International Conference on Compiler Construction*, pages 244–258. Springer-Verlag, 1999.

[82] T. Tourwé and K. De Volder. Using software classifications to drive code generation. Position paper, Programming Technology Lab, Vrije Universiteit Brussel, March 2000. ECOOP 2000 Workshop on Objects and Classification: a Natural Convergence.

[83] M. Uschold and M. Gruninger. Ontologies: Principles, methods and applications. *Knowledge Engineering Review*, 11(2), June 1996.

[84] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, August 1995.

[85] D. Westwood. *LPA WIN-PROLOG 4.0 Programming Guide*. Logic Programming Associates Ltd., 1999.

[86] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA 1998*, pages 112–124. IEEE Computer Society Press, 1998.

[87] R. Wuyts. *Logic Meta Programming as a General Approach to Support Co-evolution*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 2000. In preparation (tentative title).