Declaratively Codifying Software Architectures using Virtual Software Classifications

Kim Mens* and Roel Wuyts†
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
e-mail: { kimmens | rwuyts }@vub.ac.be

February 16, 1999

Abstract

Most current-day software engineering tools and environments do not sufficiently allow software engineers to declare or enforce the intended software architecture. On the one hand, architectures are typically described at a too low level, inhibiting their evolution and understanding. On the other hand most tools provide little support to automatically verify whether the source code conforms to the architecture. Therefore, a formalism is needed in which architectures can be expressed at a sufficiently abstract level, without losing the ability to perform automatic conformance checking. We propose to declaratively codify software architectures using virtual software classifications and relationships among these classifications. We illustrate how software architectures can be expressed elegantly in terms of these virtual classifications and how to keep them synchronized with the source code.

1 Introduction

The problem of *architectural mismatch* [GAO95] is a well-known problem: by the time an architecture specification is published, it is already wrong. It is also an important problem, because although the time to design an architecture may take up to one year, the engineers must then live with the architecture for up to 15 years of development and maintenance [SSWA96].

To solve the problem of architectural mismatch, we not only need to describe the software architecture explicitly at a sufficiently high level of abstraction, but also provide support for keeping the source code conform to it. Therefore, the contribution of this paper is to present a formalism in which architectural knowledge can be codified[1] with the ability to automatically check conformance of source code. Preferably, such a formalism should possess the following characteristics:

- It should allow *reasoning at sufficiently high levels of abstraction*, i.e., in terms of the components that are the focus of interest at that time. At architectural level, we reason about components, connectors, architectural styles, ...; at design level, we reason in terms of classes, methods, inheritance, aggregations, ... and at implementation level we are interested in individual statements, message expressions, and so on.
- We prefer a *declarative* formalism because this makes the architectural descriptions easier to understand and maintain.

*Research funded by the Brussels Capital Region (Belgium)
†Research conducted on a doctoral grant from the Instituut voor Wetenschap en Technologie (Flanders, Belgium)
[1] We use the term 'codifying' in a slightly stronger sense than [SG96]. In our terminology, 'codifying' means: "to make explicit so that it can be automatically manipulated in tools".

- Because we do not want to be limited in expressiveness, we want to use a *full-fledged programming language*. More specifically, because we want a declarative formalism, we prefer a *logic* programming language in which we can exploit the full power of *unification*.
- To allow conformance checking of the source code to the architecture, we need an explicit link between source-code artifacts and high-level architectural abstractions. We want to be able to map architectural components and relationships to all possible kinds of source-code artifacts and dependencies.
- We want the formalism to be as *open* as possible. For example, it should be possible at all times to introduce new kinds of relationships between architectural components, to reason about new kinds of source-code artifacts at architectural level, or to define new mappings between architectural components and source-level artifacts.

The formalism we propose in this paper adheres to the above characteristics. More specifically, we will introduce *virtual classifications* to map source-level artifacts to higher-level architectural components (and vice versa). We will declare *explicit relationships* between these virtual classifications that describe the co-operations among them. The underlying medium that will allow us to define virtual classifications and their relationships, and that links them to the source code is *SOUL*, the Smalltalk Open Unification Language [Wuy98]. This combination allows us to codify explicitly and declaratively software architectures at a high level of abstraction. We will present an example of declaring an architecture in this formalism, and illustrate how to automatically check conformance of the source code to this architecture.

Before explaining our approach in section 3, we highlight the essential differences with some closely related work in the next section. Section 4 illustrates and validates our approach by means of some experiments. Before concluding (section 6), section 5 discusses the achieved results and future work.

2 Related Work on Software Architectures

The architecture of a software system defines that system in terms of high-level components and interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some rationale for the design decisions [SG96]. In this paper, we focus on the structural aspects of software architectures only.

Shaw and Garlan [SG96] provide a taxonomy of some of the current issues in software architecture research. Using this taxonomy some may want to characterize our work as the search for new Architectural Description Languages (ADLs). Personally, we would situate it in the area of developing formalisms for reasoning about architectural designs. SOUL should therefore be viewed merely as a medium in which we validate our claims about the proposed formalism. We do not intend to promote SOUL as a new or better ADL.

In the area of software architecture, much research has been done that is similar to ours. Murphy [MNS95] introduces 'software reflexion models' that show where an engineer's high-level model of the software does and does not agree with a source model, based on a declarative mapping between the two models. Module Interconnection Languages (MILs) [PDN87] can be used to formally describe the global structure of a software system, by specifying the interfaces and interconnections among the components ('modules') that make up the system. These formal descriptions can be processed automatically to verify system integrity. A MIL describes the interconnections between modules in terms of the entities they contain (e.g. variables, constants, procedures, type definitions, ...). Typical syntax primitives are: *provide*, *require*, *has-access-to* and *consists-of*.

Shaw and Garlan [SG96] argue that MILs force software architects to use a lower level of abstraction than is appropriate, because they focus too much on 'implementation' rather than 'interaction' relationships between modules. In our opinion, software reflexion models suffer from

the same problem: high-level relationships between architectural components are typically mapped to calling relations, file-dependencies, cross-reference lists and so on. Our formalism tries to extend approaches such as software reflexion models or MILs, to higher levels of abstraction allowing, for example, the declaration of:

- architectural components that are mapped to multiple software artifacts spread throughout the source code;
- more complex relations dealing with transitive closures, protocols, programming conventions, design styles, ...;
- higher-level architectural components that are described in terms of other high-level components and relationships themselves.

Another difference between software reflexion models and our approach is the difference in focus. To obtain more flexibility and efficiency — at the cost of decreased precision — Gail Murphy [MN95] rejects the use of parsers, but uses lexically-based tools that produce approximate results to extract information from source code. We approach the problem from the other end of the spectrum. Because we do not want to restrict a priori the kinds of source-code artifacts we want to reason about at architectural level, we do use parsers. Furthermore, to allow powerful reasoning about this information, we use the technique of *unification*. Our goal is to allow describing software architectures at the highest abstraction level possible (without losing the ability to verify conformance of source code), at the cost of decreased efficiency.

3 Our Approach

In developing our formalism, we were inspired by De Hondt's dissertation about *software classification* as an approach to architectural recovery in evolving object-oriented systems [Hon98]. In this paper, we will investigate the power of *virtual software classifications* to codify and reason about software architectures. Another source of inspiration was SOUL, the Smalltalk Open Unification Language [Wuy98], which we chose as a medium in which to conduct our experiments. In the next subsections, we clarify both virtual classifications and SOUL, and explain which experiments we conducted to justify the claims of this paper.

3.1 Virtual Classifications

In his PhD dissertation [Hon98], De Hondt reports on positive experiences with recovering architectural knowledge in terms of simple *software classifications*. The idea of a software classification is to group together software artifacts that should be considered as a whole. All artifacts in such a classification typically share some important feature[2].

In this paper we investigate to which extent *virtual classifications* of software artifacts and the relationships among such classifications can be used to declare explicitly architectural knowledge. Virtual classifications are special classifications that *compute* their elements. This makes them more abstract than manually constructed classifications, because they actually describe in an explicit (and in our case, declarative) way which artifacts are intended to belong to the classification.

For reasons of brevity, in the remainder of this paper we will often write 'virtual classification' or even 'classification' instead of 'virtual software classification'. When we mean 'software classification' in the sense of [Hon98], this will be explicitly mentioned. Also note that our terminology of 'virtual classification' differs slightly from that of De Hondt [Hon98]: our definition of virtual classification corresponds to his definition of 'computed classification'.

[2] Jacobson [JGJ97] provides some examples of (functional) *features* and the FODA methodology [KCH+90] considers distinct types of features: operational, non-functional, development, ...

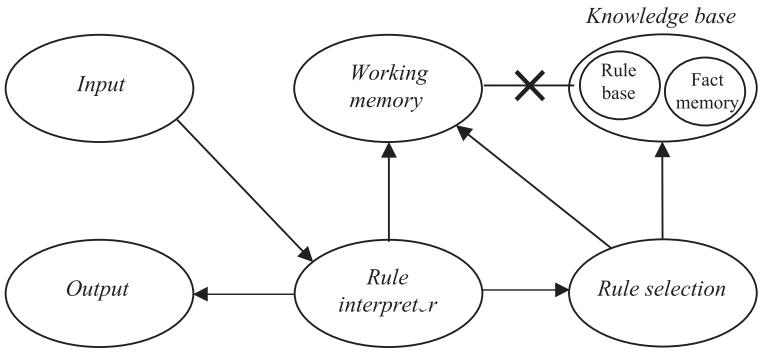

Figure 1: Rule-Based System Architecture

3.2 SOUL

SOUL is a reflective, PROLOG-like, declarative rule-based language implemented in Smalltalk [Wuy98]. It allows the declaration of structural rules about Smalltalk source code. These rules can be used to query the software system to find occurrences of certain structures, or to enforce the presence of structures. Currently, SOUL comes with a layered framework of rules that allow reasoning at the implementation and the design level.

In this paper, we not only use SOUL as a medium in which to codify architectural knowledge (by adding an architectural layer), but also as a case study to express the architecture of the SOUL reasoning engine. We chose this case, not only because of our first hand knowledge of the SOUL implementation, but also because the basic architecture of rule-based systems is explicitly documented upon in literature. Our slightly modified variant of the rule-based architecture presented in [SG96, HR85] is depicted in figure 1.

3.3 Validation

In order to validate the claims made in section 1 we need to show two things:

1. that virtual classifications and their relationships codify software architectures at a sufficiently abstract level,

2. that we can automatically check conformance of source code to the architectural descriptions.

Therefore, we set up the following experiment: we codify the architecture of the SOUL reasoning engine and check the conformance of the actual (Smalltalk) implementation of SOUL to this architecture. The architecture is depicted in figure 2 which is a refinement of figure 1. More particularly the names of some virtual classifications were made a bit more specific, the relationships were given an intuitive name such as *uses* or *creates* (optionally annotated with a '*' denoting the transitive closure of that relationship), and two cardinalities were attached to each relationship. The cardinality $\exists$ means 'one or more' or 'at least one' and $\forall$ means 'every'. For example the 'uses' relationship between 'Input' and 'Query Interpreter' could be read as: 'every item in the Input classification should use at least one item in the Query Interpreter classification'. Note that the *all*-to-one cardinality $\forall$ is stronger than typical many-to-one cardinalities available in most design notations.

4 Experiments

Using the experiment described in section 3.3 we argue why virtual classifications and their relationships seem to provide a good abstraction for describing architectural entities, ranging from

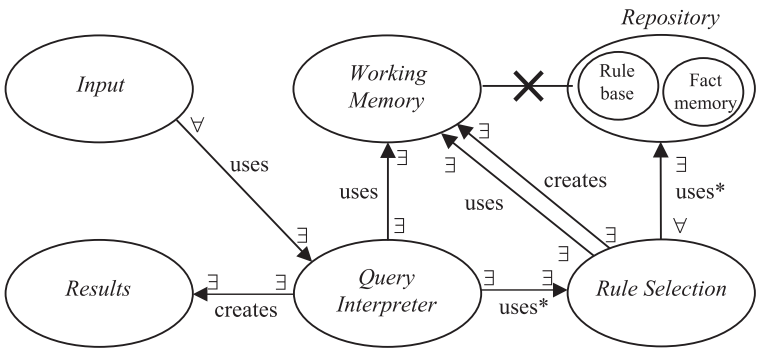

Figure 2: SOUL Reasoning Engine Architecture

architectural components and connectors, through architectures and subarchitectures, to architectural patterns. Using the same experiment we explain how to perform conformance checking: we explain how virtual classifications can be computed from source-code artifacts, how relations among virtual classifications are mapped to more primitive dependencies between source-code artifacts and how to check conformance of entire architectures (possibly containing subarchitectures).

4.1 Declaring Virtual Classifications

This first section declares the virtual classifications of the SOUL reasoning engine architecture depicted in figure 2. This illustrates how virtual classifications expressed in SOUL provide an abstract mapping of high-level architectural entities to low-level source-code artifacts. All code extracts are presented in SOUL syntax which is very similar to Prolog syntax, with a few noteworthy differences. Instead of using square brackets '[' and ']', SOUL lists are delimited by '<' and '>'. In SOUL, '[...]' denotes a re-ification of Smalltalk code in the SOUL language. Finally, SOUL variables start with a '?' instead of with a capital.

The experiment starts with virtual classifications that are straightforward mappings to source-code artifacts, but we then show mappings that cross-cut the system, or that are expressed in terms of other architectural entities.

Working Memory. The first virtual classification we declare is *workingMemory*. Conceptually this classification contains only those classes that have something to do with the working memory of the SOUL reasoning engine. This boils down to the *SOULBindings* class and all subclasses thereof, because these are responsible for handling the bindings that are assigned to variables during unification. This is straightforwardly implemented as follows, using the predefined *hierarchy* predicate:

```
Rule classIsClassifiedAs(?Class, workingMemory) if
        hierarchy([SOULBindings], ?Class).
```

Once this rule is defined, we can launch a query that computes all classes belonging to this classification. This query results in every class that is a subclass of *SOULBindings*.

```
Query classIsClassifiedAs(?Class, workingMemory).
```

Rule Selection. The classification *ruleSelection* groups all software artifacts that deal with selecting the relevant rules from the logic repository during query evaluation. This classification consists of methods[3] only, namely all methods named *unifyingClauses:* defined on classes in

[3] In SOUL, methods will be represented by their parse-tree representation, including their class, name and body.

the SOUL system. The body of the rule implementing this consists of a conjunction of two statements: the first restricts the scope to SOUL classes (using an auxiliary classification *soulClass*), and the second retrieves from these classes all methods named *unifyingClauses:*. The predefined SOUL predicate *classImplements* finds the method with a given name in a given class.

Rule methodIsClassifiedAs(?Method, ruleSelection) **if**
classIsClassifiedAs(?Class, soulClass),
classImplements(?Class, [#unifyingClauses:], ?Method).

Although this classification looks very simple, it actually specifies a mapping that crosses the boundaries among many different classes and hierarchies: the *ruleSelection* methods can be spread throughout the entire SOUL system.

Query Interpreter. The *queryInterpreter* classification consists of all methods that deal with the actual interpretation of queries. Conceptually, these are all the methods that get called — directly or indirectly — when a query is interpreted. Because interpretation of a query is started by invoking the method *interpret:repository:* on class *SOULQuery*, we merely need to compute the transitive closure of all methods that are invoked by this method. For reasons of efficiency, we also restrict the scope to relevant classes and methods only, skipping for example methods that have to do with input/output, displaying, ... All this is done by the auxiliary predicate *reaches*.

Rule methodIsClassifiedAs(?Method, queryInterpreter) **if**
classImplements([SOULQuery], [#interpret:repository:], ?M),
reaches(?M, ?Method).

Note that this classification defines a real cross-cut of the SOUL code, starting from one method and collecting all methods that are transitively invoked by this initiating method.

Input. The *input* classification is an example of a classification that is not defined directly as a mapping to lower-level artifacts, but at a higher and more abstract level, in terms of its relation with another classification. It contains the classes that initiate the interpretation process of a query. Conceptually, these are the classes that belong to the *soulApplication* classification consisting of all SOUL GUI applications, and that use a method that is classified in the *queryInterpreter* classification.

Rule classIsClassifiedAs(?Class, input) **if**
classIsClassifiedAs(?Class, soulApplication),
methodIsClassifiedAs(?Method, queryInterpreter),
uses(?Class, ?Method).

Repository. The *repository* of a rule-based system is typically made up of a rule base and the fact memory. We defined this classification as consisting of methods that access (read or write) directly or indirectly the instance variable *clauses* of class *SOULRepository*.

Results. The *results* classification contains software artifacts that deal with the results of queries. In SOUL this is done by one class, *SOULResult*, so the classification contains just this class and its subclasses.

4.2 Codifying a Software Architecture

This section illustrates how to codify a software architecture using virtual classifications and relationships among them. We also show how conformance checking from source code to an architecture is done.

4.2.1 Relationships between Components

Before discussing how to describe software architectures using virtual classifications and their relationships, we first explain the kinds of relationships that can be expressed. The idea is to connect architectural components (in our case: virtual classifications) with high-level intuitive connectors such as *uses*, *creates*, *accesses*, ... However, because we want to check conformance to architectures, and architectural relationships in particular, we need to map these high-level connectors to more primitive dependencies (such as message invocations, instance creation, reading or writing variables, and combinations thereof) that can actually be found in the source code.

In order to map a relationship between 2 classifications containing many source-code artifacts to dependencies between those artifacts, we also need to specify cardinalities. For example,

uses(allToOne,input,queryInterpreter)

means that *all* artifacts in the *input* classification should *use* at least *one* artifact in the *queryInterpreter* classification. Other cardinalities are *oneToOne*, *oneToAll*, and *allToAll*. In the examples that follow, we will often use shortcuts such as

usesAllToOne(input,queryInterpreter)

where the cardinalities are absorbed in the name of the predicate instead of given as extra argument when calling the predicate.

The *uses* relationship (and others) is not only defined between classifications, but is overloaded at many levels of abstraction. At the highest abstraction level, it works with classification names. This is translated into a *uses* relationship between lists of source-code artifacts (corresponding to the classifications named). Next, using the specified cardinality, this is translated to one or more *uses* relationships among the classified artifacts. Depending on the kinds of artifacts, the relationship is further refined. In the case of *uses* between two methods, we just check whether there is a message invocation between the two. When one of the arguments is a class, we define the *uses* relationship in terms of a *uses* relationship on the methods of that class. For example, a class uses a method if at least one of its methods uses that method.

Finally, we want to stress that the high-level connectors between architectural components can have an arbitrary complexity. *uses* is an example of a simple connector that maps almost directly to message invocations at source-code level. The *creates* relationship is a bit more complex. Without going into the details, the mapping for this relationship takes into account both lazy initialization and direct invocation of class creation methods (constructors) [Bec97], and instance creation through factory methods or class factories [GHJV94]. Another example of a more complex relationship is *usesTrans* which corresponds to the transitive closure of the *uses* relationship. Also negative relationships (stating for example that two classifications should *not* be connected) can be expressed.

4.2.2 The SOUL Architecture

As an example of a concrete architecture, we illustrate how the architecture of the SOUL reasoning engine (figure 2) can be expressed in terms of the virtual classifications declared in section 4.1. As is illustrated by the declarations below, an architectural description consists of a unique name, a list of components of which the architecture is composed (in this example, the components are virtual classifications), and a list of relationships among the architectural components.

Fact architecture(soul,
< input, queryInterpreter, workingMemory,
ruleSelection, repository, results >,
< usesAllToOne(input, queryInterpreter),
usesOneToOne(queryInterpreter, workingMemory),
usesTransOneToOne(queryInterpreter, ruleSelection),
usesTransAllToOne(ruleSelection, repository),
createsOneToOne(ruleSelection, workingMemory),
usesOneToOne(ruleSelection, workingMemory),

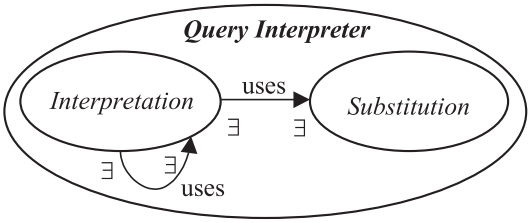

Figure 3: Query Interpreter Architecture

createsOneToOne(queryInterpreter, results),
not(related(workingMemory, repository)) >).

4.2.3 Conformance Checking

Once an architecture description has been asserted, it can be used to verify whether the source code conforms to it. Indeed, because an architecture declares relationships among virtual classifications, and virtual classifications and their relationships can be mapped to source-code artifacts and dependencies among these artifacts, an architecture indirectly defines a relationship between source-code artifacts. To check whether an architecture with some name is valid (i.e., whether the source code conforms to it), we

1. fetch the description of the architecture with that name,

2. check whether the architecture is well-formed, i.e., that the components have been declared and that the relationships between the enumerated components are well-formed,

3. check whether the declared relationships between the architectural components hold.

Rule checkArchitecture(?ArchitectureName) **if**
architecture(?ArchitectureName, ?Components, ?Relationships),
wellFormedArchitecture(?Components, ?Relationships),
checkArchitecturalRelationships(?Relationships).

The auxiliary predicate *checkArchitecturalRelationships* checks whether the declared relationships hold, by invoking them one by one. For example, in the case of the *soul* architecture, the first relationship

usesAllToOne(input,queryInterpreter)

checks whether every item in the input classification uses at least one item in the *queryInterpreter* classification (see also section 4.2.1).

Our initial conformance checking tool was very primitive, merely returning a 'true' or 'false' depending on the success of the conformance check. We are currently extending the tool to provide more detailed information on the conformance checking process. More precisely, in analogy to [MNS95], we could compute the *convergences* (where the source code agrees with the architecture), the *divergences* (where the source code shows dependencies that are not predicted by the architecture) and the *absences* (where the source code does *not* contain dependencies that are described by the architecture). We do this by making our cardinality predicates (∀ and ∃) more intelligent. For example, when checking a relationship, the predicate for ∃ remembers for which artifacts the relationship holds and the predicate for ∀ remembers which artifacts failed to satisfy the relationship (if any). However, because the predicates are implemented with lazy evaluation — for efficiency reasons —, the extra information they provide is restricted by their lazyness.

4.3 Refining a Classification as a Software Architecture

The *soul* architecture codified in section 4.2.2 used only software classifications as components. However, software architectures can have subarchitectures as components as well. To illustrate this, this section refines the *queryInterpreter* classification presented earlier as an architecture itself. More precisely, we define two subclassifications and declare relationships between them, as illustrated in figure 3. This results in a classification *queryInterpreter* that is defined at a high level of abstraction, and that can still be checked for source-code conformance.

4.3.1 The Interpretation and Substitution Subclassifications

The original *queryInterpreter* classification consisted of software artifacts that deal with the interpretation of queries. This interpretation process actually consists of two phases: the interpretation phase where terms and clauses are interpreted, and a substitution phase, where bindings found during unification are substituted in the term that is currently being interpreted. We will declare these two phases as subclassifications: *interpretation* and *substitution*. Note that both implementations use Smalltalk *protocols*[4] to select the relevant methods.

Rule methodIsClassifiedAs(?Method, interpretation) **if**
classIsClassifiedAs(?Class, soulClass),
or(methodInProtocol(?Class, [#interpretation], ?Method),
methodInProtocol(?Class, [#interpreting], ?Method)).

Rule methodIsClassifiedAs(?Method, substitution) **if**
classIsClassifiedAs(?Class, soulClass),
methodInProtocol(?Class, [#substitution], ?Method).

4.3.2 The QueryInterpreter Subarchitecture

Based on these subclassifications we can redefine the *queryInterpreter* classification as their union.

Rule isClassifiedAs(?artifact, queryInterpreter) **if**
unionClassification(interpretation, substitution, ?C),
member(?artifact, ?C).

However, because there are a number of important relationships between these subclassifications, the ruleInterpreter classification is more than a mere union. In fact, it is in turn an architecture composed of these subclassifications together with their relationships.

Fact architecture(queryInterpreter,
< interpretation, substitution >,
< usesOneToOne(interpretation, substitution),
usesOneToOne(interpretation, interpretation) >).

When adopting a coarse-grained view, *queryInterpreter* is simply a classification that is connected to other classifications. But in a more fine-grained view, we consider it as an architecture consisting of several components with their own relationships. In the latter view the connections between *queryInterpreter* and other classifications can be refined in terms of these components. This is typically done by introducing *ports* [SG96]. We simulate ports by defining a port mapping that refines relationships between architectural components in terms of relationships between their subcomponents. For example, we can refine the *uses* relationship from *input* to *queryInterpreter*, into a *uses* relationship from *input* to *interpretation* (which is a subclassification of *queryInterpreter*). The mapping for this example is implemented by the first fact below. Figure 4 illustrates

[4] Smalltalk environments typically subdivide the methods of a class in protocols such as *printing*, *accessing*, *initialize*, ...

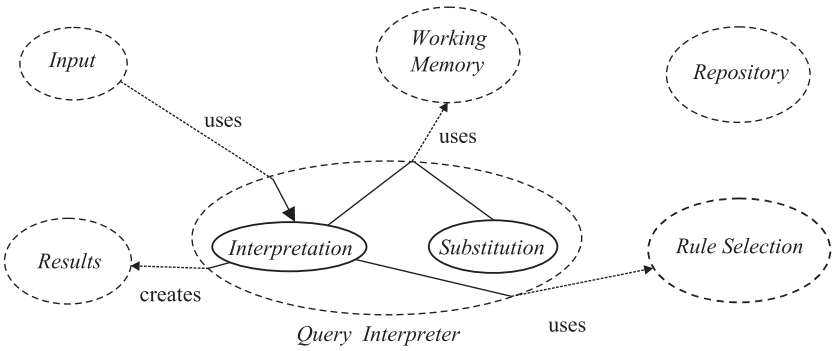

Figure 4: Port Mapping

some other mappings, and their SOUL implementation is also presented below. Note that some relationships may be refined into more than one relationship, as is the case for the *uses* relationship between *queryInterpreter* and *workingMemory*.

Fact portMapping(uses, input, queryInterpreter, input, interpretation).
Fact portMapping(creates, queryInterpreter, output, interpretation, output).
Fact portMapping(uses, queryInterpreter, workingMemory,
interpretation, workingMemory).
Fact portMapping(uses, queryInterpreter, workingMemory,
substitution, workingMemory).
Fact portMapping(uses,queryInterpreter,ruleSelection,
interpretation,ruleSelection).

For some relationships between classifications, a mapping in terms of their subclassifications may not be provided. When checking those relationships, the coarse-grained view is adopted, by considering the classifications as a mere union of their subclassifications.

4.3.3 Conformance Checking — Revisited

Finally, we revisit the conformance checking rules in the current situation where classifications can be architectures that are, in turn, built up from many classifications. To deal with this situation, the *checkArchitecture* rule needs to be refined with an extra clause *checkSubArchitectures(?Components)* which checks for each of the components whether its architecture is valid. For components that are ordinary classifications, nothing special needs to be done, but for components that are again architectures, the *checkArchitecture* predicate is called recursively to check conformance to that (sub)architecture. Also, the *checkArchitecturalRelationships* predicate needs to take the port mappings into account. For those relationships that are refined by a port mapping, the refined version(s) should be checked.

Rule checkArchitecture(?ArchitectureName) **if**
architecture(?ArchitectureName, ?Components, ?Relationships),
wellFormedArchitecture(?Components, ?Relationships),
checkSubArchitectures(?Components),
checkArchitecturalRelationships(?Relationships).

4.4 Architectural Patterns

This section shows how we can easily define *architectural patterns* in our formalism, demonstrating again that it can be used at very high levels of abstraction, without losing the ability to do

conformance checking. As an example we define the *rule-based system* architectural pattern (see figure 1), of which the *soul* architecture is a specific instance.

An architectural pattern describes an architectural structure consisting of architectural components and relationships. However, as opposed to a concrete architecture, it provides a *template* that can contain 'holes' (implemented by logic variables) that need to be filled in upon instantiation. The predicates below implement the rule-based system architectural pattern, and show how this template can be used to re-implement the *soul* architecture in term of this pattern.

```
Fact ruleBasedSystemPattern(
        ?Related1, ?Related2, ?Related3, ?Related4, ?Related5, ?Related6,
        description( <  ?Input, ?RuleInterpreter, ?WorkingMemory,
                        ?RuleSelection, ?KnowledgeBase, ?Output  >,
                     <  ?Related1(?Input, ?RuleInterpreter),
                        ?Related2(?RuleInterpreter, ?WorkingMemory),
                        ?Related3(?RuleInterpreter, ?RuleSelection),
                        ?Related4(?RuleSelection, ?KnowledgeBase),
                        ?Related5(?RuleSelection, ?WorkingMemory),
                        ?Related6(?RuleInterpreter, ?Output),
                        not(related(?WorkingMemory, ?KnowledgeBase))  >)).

Rule architecture(soul, ?Components, ?Relationships) if
        equals(?Components,
                     <  input, queryInterpreter, workingMemory,
                        ruleSelection, repository, results  >),
        ruleBasedSystemPattern(
                     usesAllToOne, usesOneToOne, usesTransOneToOne,
                     usesTransAllToOne, createsAndUsesOneToOne, createsOneToOne,
                     description(?Components, ?Relationships)).
```

Note that the rule instantiating the *soul* architecture merely fills in the holes of the pattern by supplying the concrete components and relationships. Although not visible in this example, upon instantiation of a pattern, some more specific constraints that are not provided by the pattern may be declared as well. Similarly, more complex architectural patterns make use not only of (unification of) logic variables, but also use logic reasoning to declare the structure of the pattern itself (e.g., the relationships between the components). An example of such a pattern is the *pipe and filter* architectural pattern defined below.

```
Rule pipeAndFilterPattern(?Connector, description(?Filters, ?Pipes)) if
        computePipes(?Connector, ?Filters, ?Pipes).
```

The *computePipes* auxiliary predicate states that the set *?Pipes* of relationships between *?Filters* should be such that the first filter is connected to the second one, the second to the third one, and so on. The variable *?Connector* denotes the kind of connection.

```
Rule computePipes( ?Connector, < ?Filter1, ?Filter2 | ?OtherFilters >,
                   < ?Connector(?Filter1, ?Filter2) | ?OtherPipes >) if
        computePipes(?Connector, < ?Filter2 | ?OtherFilters >, ?OtherPipes).

Fact computePipes(?KindOfPipe, <?LastFilter >, <>).
```

5 Discussion and Future Work

Our initial experiments were very promising. We actually succeeded in declaring the architecture of part of the SOUL system, and checking conformance of the source code to this architecture. We

even defined subarchitectures and declared the architecture in terms of an architectural pattern, while still being able to check conformance.

To validate the practical usability (efficiency, simplicity, ease of use, readability, ...) of our formalism, more experiments are needed. One such experiment is to declare an architecture and check conformance of different versions of the source code to it. E.g., what should we do when a new version no longer conforms to the architecture? A related experiment is to investigate how refining an architecture affects the source code's conformance to it. Experiments such as these fit in our longer term research goal to develop a formalism for managing the evolution of software architectures. To achieve this goal, we will extend the current formalism with the technique of *reuse contracts* [SLMD96, Luc97]. The reuse contract technique facilitates software reuse and evolution of software artifacts by explicitly annotating reused or modified artifacts, as well as the modifications themselves, by so-called "reuse contracts". Reuse contracts extend existing notations with extra information on the hidden assumptions of software artifacts and the modifications made to them. This information allows us to (semi-automatically) detect potential evolution conflicts, and gives an indication of where and why these conflicts occur.

During the initial experiments, we noticed that the current implementation of our formalism is not very performant. Computing some classifications or checking some relationships (especially those involving transitive closures) can take a very long time (more than one hour). Therefore, in the near future we want to incorporate in SOUL extra search techniques and advanced unification schemes to gain more performance.

Another future research track is to further develop and promote SOUL as a general medium for expressing *software development styles* ranging from programming conventions and idioms [Mic97], through design patterns [Wuy98], to software architectures and architectural styles.

6 Conclusion

The experiments showed that our consistent use of a declarative programming language throughout all abstraction layers — from source-code level through the design and architectural level to architectural patterns — provides a viable formalism to reason about architectural knowledge on a sufficiently high level of abstraction while still allowing conformance checking of source code. Virtual classifications proved their worth as suitable abstractions of architectural components. They hide the details of the lower-level design and source-code artifacts on which they are mapped, yet allowing us to reason about their relationships with other architectural components independently of the artifacts they actually contain. Our layered formalism also provides a powerful way of defining highly abstract relationships between architectural components, 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. Finally, these mappings of architectural components to lower-level components, and of architectural relationships to lower-level relationships, made it easy to implement the conformance checking rules by implementing conformance checking at a high level in terms of conformance checking rules at lower levels, all the way down to the source code.

7 Acknowledgements

We wish to thank our advisor Theo D'Hondt for the many fruitful discussions which led to this paper. Furthermore we thank our colleagues at the Programming Technology Lab for proofreading and discussing this paper: Kris De Volder, Carine Lucas, Tom Mens, Tom Tourwé and Bart Wouters.

References

[Bec97] K. Beck.
Smalltalk Best Practice Patterns.
Prentice Hall, 1997.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom.
Architectural mismatch: Why reuse is so hard.
IEEE Software, November 1995.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.
Design Patterns.
Addisson-Wesley, 1994.

[Hon98] K. De Hondt.
A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems.
PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, 1998.

[HR85] F. Hayes-Roth.
Rule-based systems.
Communications of the ACM, 28(9):921–932, September 1985.

[JGJ97] I. Jacobson, M.L. Griss, and P. Jonsson.
Software Reuse: Architecture, Process and Organization for Business Success.
Addisson-Wesley, 1997.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson.
Feature-oriented domain analysis (foda) feasibility study.
Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[Luc97] Carine Lucas.
Documenting Reuse and Evolution with Reuse Contracts.
PhD thesis, Dept. of Computer Science, Vrije Universiteit Brussel, Belgium, 1997.

[Mic97] I. Michiels.
Using logic meta-programming for building sophisticated development tools.
Computer science graduation report, Vrije Universiteit Brussel, Belgium, 1997.

[MN95] G. Murphy and D. Notkin.
Lightweight source model extraction.
In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 116–127. ACM Press, 1995.

[MNS95] G. Murphy, D. Notkin, and K. Sullivan.
Software reflexion models: Bridging the gap between source and high-level models.
In *Proceedings of SIGSOFT'95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[PDN87] R. Prieto-Diaz and J. M. Neighbors.
Module interconnection languages.
Journal of Systems and Software, 6(4):307–334, November 1987.

[SG96] M. Shaw and D. Garlan.
Software Architecture — Perspectives on an Emerging Discipline.
Prentice Hall, 1996.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt.
Reuse contracts: Managing the evolution of reusable assets.
In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285. ACM Press, 1996.

[SSWA96] R. W. Schwanke, V. A. Strack, and T. Werthmann-Auzinger.
Industrial software architecture with gestalt.
In *Proceedings of IWSSD-8.* IEEE, 1996.

[Wuy98] R. Wuyts.
Declarative reasoning about the structure of object-oriented systems.
In *Proceedings TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.