# Declaratively Codifying Software Architectures Using Virtual Software Classifications

Kim Mens, Roel Wuyts and Theo D'Hondt
Programming Technology Lab, Vrije Universiteit Brussel
{ kimmens | rwuyts | tjdhondt }@vub.ac.be

**Abstract**

*Most current-day software engineering tools and environments do not sufficiently support software engineers to declare or to enforce the intended software architecture. Architectures are typically described at a too low level, inhibiting their evolution and understanding. Furthermore, most tools provide little support to verify automatically 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 conformance checking automatically. We propose to codify declaratively 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* [3] 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 designing an architecture may take up to one year, the engineers must then live with the architecture for up to fifteen years of development and maintenance [12]. A related problems is the problem of *architectural erosion*. Perry and Wolf [10] define architectural erosion as "violations in the architecture that lead to increased system problems and brittleness". Because the implementation of a software application continually evolves, most software applications tend to drift away from their original architecture, eventually turning the software into a legacy system when no proper actions are taken.

To counter these problems, we not only need to describe software architectures explicitly at a sufficiently high level of abstraction, but also provide support for keeping the source code conform to it. The contribution of this paper is to present a formalism in which architectural knowledge can be codified with the ability to check conformance of source code automatically. 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 concepts that are the focus of interest at that time. At architectural level, we reason about components, connectors, architectural patterns, ... 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.

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

Although our ultimate goal is to develop a tool or environment supporting declaration and verification of architectures, in this paper we mainly focus on the formalism underlying such a tool. The formalism adheres to the above characteristics. We use *virtual classifications* to map high-level architectural components to source-level artifacts (and vice versa). We declare *explicit relationships* between these virtual classifications to describe the co-operations among them. The medium in which we define virtual classifications and their relationships, and that links them to the source code is *SOUL*, the Smalltalk Open Unification Language [15]. We will present an example of declaring an architecture at a high level of abstraction in this medium, and illustrate how to check conformance of the source code to this architecture automatically .

Before explaining our approach in section 3, we discuss some closely related work in the next section. Sections 4 to 7 illustrate our formalism by means of some experiments. Before concluding (section 9), section 8 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 also provides some rationale for the design decisions in terms of the system requirements [13]. In this paper, however, we focus on the structural aspects of software architectures only.

Much research has been done on *architecture description languages* (ADLs) [13, 9]. ADLs provide a formal basis for describing software architectures by specifying the syntax and semantics of modeling components, connectors, and configurations. Although some would characterize our work as the search for a new ADL, personally, we would situate it in the area of developing formalisms for reasoning about architectural designs. In this paper, we do not intend to promote our formalism as a better or more general ADL. We merely want to validate our claim that virtual classifications are a useful abstraction for codifying software architectures.

Murphy's work on *software reflexion models* [8] is closely related to ours. Software reflexion models 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) [11] can be used to describe formally the global structure of a software system, by specifying the interfaces and interconnections among the system modules. 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, ...).

Shaw and Garlan [13] 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 relationships dealing with transitive closures, protocols, programming conventions, design styles, ...;
- higher-level architectural components and relationships 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 — Murphy [7] rejects the use of parsers, but uses lexically-based tools that produce approximate results when extracting 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 on *software classification* as an approach to architectural recovery in evolving object-oriented systems [5]. De Hondt presents the *software architecture model* — where classifications are containers of software artifacts, and artifacts can be classified in multiple classifications — as a powerful model to organize software artifacts in a flexible and uniform manner. He uses these software classifications to capture architectural abstractions that were reverse engineered from lower-level software artifacts and their interrelationships.

Whereas De Hondt distinguishes several kind of software classifications, in this paper we investigate to which extent *virtual software classifications* and their interrelationships can be used to codify software architectures and to reason about them. Virtual classifications are special classifications that can *compute* their elements. This makes them more abstract than, for example, manually constructed classifications, because they 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'.

Another source of inspiration was *SOUL*, the Smalltalk Open Unification Language [15], which we chose as a medium in which to conduct our experiments. SOUL is a reflective,

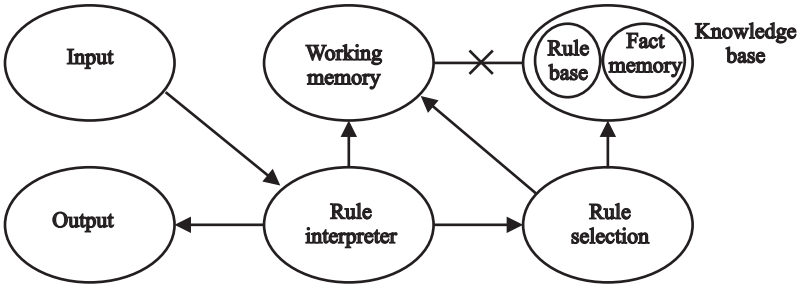


**Figure 1. Rule-based system architecture**

PROLOG-like, declarative rule-based language implemented in Smalltalk. Smalltalk was chosen as base language because of its very good reflective facilities which make processing source code much easier. Building on these facilities, SOUL 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 rule engine. We chose this particular 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. Another reason for choosing this case it that it allows us to explain both the architecture of the SOUL system and some of the principles behind SOUL at the same time. Our slightly modified variant of the rule-based architecture presented in [13, 4] is depicted in figure 1.

To validate our claims, we set up the following experiment: we codify the architecture of the SOUL rule engine and check conformance of the Smalltalk implementation of SOUL to this architecture. The codified 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 '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.

Using this experiment we argue why virtual classifications and their relationships provide a good abstraction for describing architectural entities, ranging from architectural components and connectors, through architectures and subarchitectures, to architectural patterns. Using the same experiment we explain how to perform conformance checking of source code to architectures: 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).

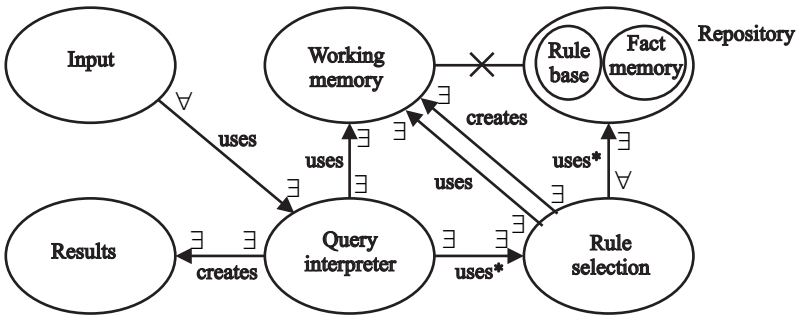


**Figure 2. SOUL rule engine architecture**

## 4 Declaring virtual classifications

In this section we declare the virtual classifications of the SOUL rule 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. Also note that, for example, methods of the same class can be classified in many different classifications.

**Working memory.** Conceptually, the *workingMemory* classification contains only those classes that have something to do with the working memory of the SOUL rule engine. In the source code, 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, 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 succeeds for 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 upon query evaluation. This classification consists of methods[1] only, namely all methods named *unifyingClauses:* defined on classes in 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.

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

```
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 belong to many classes spread throughout the entire SOUL implementation.

**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 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 an auxiliary predicate *reaches*.

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

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. This classification merely contains the class *SOULResult* and its subclasses.

## 5 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.

### 5.1 Relationships between components

Before discussing how to describe software architectures, we 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* and

*accesses*. However, because we want to check source-code 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 two 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 used in this paper 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 (among 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 groups 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) [1], and instance creation through factory methods or class factories [2]. 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.

### 5.2 The SOUL architecture

As an example of a concrete architecture, we illustrate how the architecture of the SOUL rule engine (figure 2) can be expressed in terms of the virtual classifications declared in section 4. As is illustrated by the declarations below, an architecture 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),
```

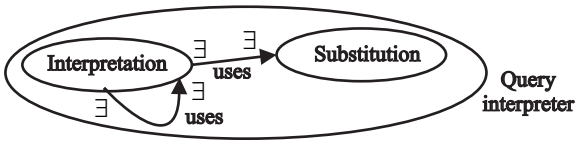


**Figure 3. Query interpreter subarchitecture**

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

### 5.3 Conformance checking

Once an architecture has been declared, 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 first fetch the description of the architecture with that name. Then, we 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. Finally, we check whether these 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 5.1).

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 [8], 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 can do this by making our cardinality predicates ($\forall$ and $\exists$) more intelligent. For example, when checking a relationship, the predicate for $\exists$ could remember for which artifacts the relationship holds and the predicate for $\forall$ could remember 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.

## 6 Refining a classification as a software architecture

The *soul* architecture codified in section 5.2 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 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.

### 6.1 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*.

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

Note that both rules use Smalltalk *protocols* to select the relevant methods. Smalltalk environments typically subdivide the methods of a class in protocols such as *printing*, *accessing*, *initialize*, ... Although we can reason at a more semantic level by relying on such information, we are dependent on the developers' goodwill to use these protocols in a disciplined way. However, the task of filling in these protocols (or other tags) could be partially supported by a tool integrated in the software development environment. In languages other than Smalltalk, protocols can be simulated by allowing software engineers to explicitly 'tag' software artifacts with extra semantic information during the development process.

### 6.2 The queryInterpreter subarchitecture

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

```
Rule isClassifiedAs(?Artifact,queryInterpreter) if
        union(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 (see figure 3).

```
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 subcomponents with their own relationships. In the latter view the connections between *queryInterpreter* and other classifications can be refined in terms of these subcomponents. This is typically done by introducing *ports* [13]. We simulate ports by defining *port bindings* that refine relationships between architectural components

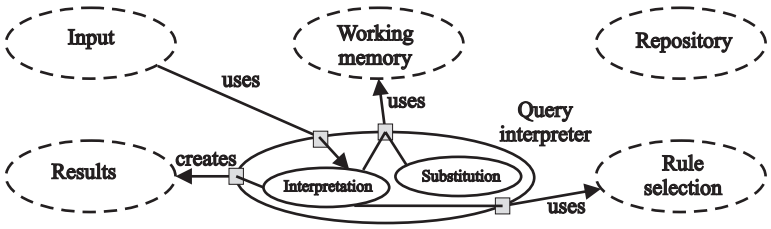


**Figure 4. Port bindings**

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 port binding for this example is implemented by the first fact below. Figure 4 illustrates some other bindings, and their SOUL implementation is given 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 portBinding( uses,input,queryInterpreter,input,interpretation).
Fact portBinding( creates,queryInterpreter,results,interpretation,results).
Fact portBinding( uses,queryInterpreter,workingMemory,
                  interpretation,workingMemory).
Fact portBinding( uses,queryInterpreter,workingMemory,
                  substitution,workingMemory).
Fact portBinding( uses,queryInterpreter,ruleSelection,
                  interpretation,ruleSelection).
```

For some relationships between classifications, no port binding may be provided. When checking those relationships, the coarse-grained view is adopted, by considering the classifications as a mere union of their subclassifications.

### 6.3 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 bindings into account. For those relationships that are refined by a port binding, the refined version(s) should be checked.

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

## 7 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 makes heavy use of logic unification when filling in the holes of the pattern with the concrete components and relationships. Although not present 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). Due to space limitations, we could not include an example illustrating this.

## 8 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 a subarchitecture and declared an architecture in terms of an architectural pattern, while still being able to check conformance. However, as one single case study is not sufficient to prove our claims, more case studies will be performed in the near future.

The architectural model employed in this paper was rather primitive. For example, we only considered fairly simple connectors such as *uses* and *creates* that can be mapped rather straightforwardly to lower-level implementation relationships. We are currently extending the model to deal with higher-level connectors or even connectors constructed from other connectors themselves. Preferably, we want architectural connectors that include more

semantics than *uses* or *creates*, for example, by relying on particular conventions adopted by the engineers, or by reasoning about specific tagging information annotated by the engineers during development. Another extension is to allow other (perhaps user-defined) kinds of cardinalities.

To validate the practical usability (efficiency, simplicity, ease of use, readability, ...) of our formalism, more experiments are needed as well. One such experiment is to declare an architecture and check conformance of different versions of the source code to it. 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 unanticipated evolution of software architectures. To achieve this goal, we will extend the current formalism with the technique of *reuse contracts* [14, 6], which is specifically targeted for dealing with unanticipated evolution of software artifacts (and automated conflict checking in particular).

During our 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 (sometimes more than one hour). To gain more performance, we want to extend SOUL with extra search techniques and more advanced unification schemes.

The current formalism is only supported by a very primitive tool. Therefore, in parallel with optimizing and extending the formalism, a real tool or environment supporting declaration and verification of architectures should be developed.

Another future research track is to develop SOUL further and to promote SOUL as a general medium for expressing *software development styles* ranging from programming conventions and idioms, through design patterns [15], to software architectures and architectural patterns.

## 9 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.

## 10 Acknowledgements

We wish to thank our colleagues at the Programming Technology Lab for proof-reading and discussing early versions of this paper: Kris De Volder, Carine Lucas, Tom Mens, Tom Tourwé and Bart Wouters. We are also grateful to Serge Demeyer, Patrick Steyaert and the anonymous referees for commenting on a more final version of this paper.

Kim Mens' research is funded by the Brussels Capital Region (Belgium) and Wang Global. Roel Wuyts' research is conducted on a doctoral grant from the "Instituut ter bevordering van het Wetenschappelijk en Technologisch onderzoek in de industrie" (Flanders, Belgium).

## References

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

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

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

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

[5] 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.

[6] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts.* PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, September 1997.

[7] 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.

[8] 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.

[9] 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.

[10] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

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

[12] 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.

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

[14] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of the OOPSLA'96 Conference on Object-Oriented Programming, Systems, Languages and Applications*, number 31(10) in ACM SIGPLAN Notices, pages 268–285. ACM Press, 1996.

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