Synchronising Changes to Design and Implementation using a Declarative Meta-Programming Language

Roel Wuyts
roel.wuyts@iam.unibe.ch
Software Composition Group
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

Abstract

When developing software systems, the relation between design and implementation is typically left unspecified. As a result design or implementation can be modified independently of each other, and a modification of either one does not leave any trace in the other. The practical result of this is a number of well-known problems such as drift and erosion, documentation maintenance problems or round-trip engineering trouble. To solve these problems we propose to make the relation between design and implementation explicit by expressing design as a *logic meta-program* over implementation. This is the cornerstone for building a complete synchronisation framework that allows one to synchronise changes to design and implementation. We have implemented such synchronisation framework, and applied it successfully on two case studies.

1 Introduction

One of the main problems in software engineering we see is that the relation between design and implementation is typically left implicit, and almost never explicitly gets recorded. The result is a number of well-known problems, such as design and implementation drifting apart during development, or the documentation quickly getting out of sync with respect to the implementation. In this paper we tackle this problem with a solution based on the following cornerstones:

1. express design as a logic meta-program over implementation: hence the relation is made explicit in a full-fledged programming language;

2. use the logic meta-programming language as the synchronisation engine and to define actions. Because the mapping is expressed as a logic meta-program, we can run logic programs that compare design and implementation and implement actions when differences are found (such as adding or removing items in either design or implementation);

3. integrate in the development environment to receive notifications of changes: when the development environment is able to notify the synchronisation engine whenever changes are made to design or implementation, the synchronisation engine can guide design or development (by indicating design violations, or updating the design as needed, etc.)

We implemented this solution as a framework (the *synchronisation framework*), that allows to synchronise changes between design and implementation. Its general structure is shown in Figure 1. The main participants are the ones found in the conceptual solution outlined above: the *Declarative Framework* that relates design and implementation, a logic meta-programming language that is used as synchronisation engine and the *design and implementation monitors* that integrate the synchronisation engine in the development environment.

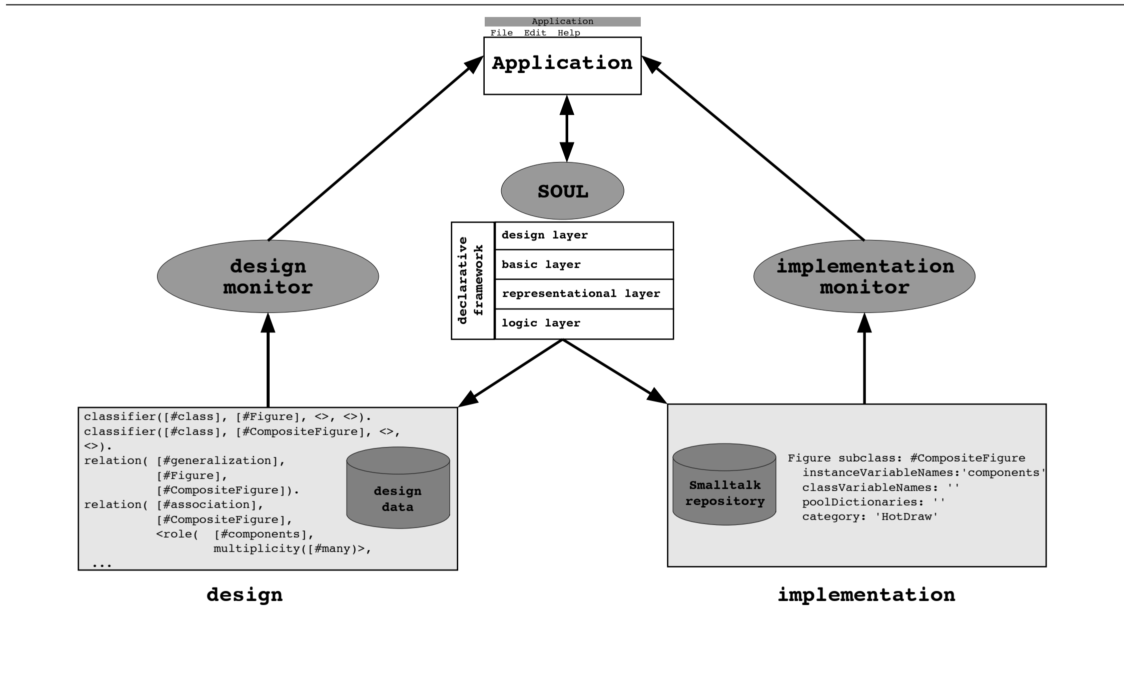

Figure 1: The general setup of the synchronisation framework*, showing the main constituents: the* logic meta-programming language SOUL*, the* declarative framework *and the* design and implementation monitors*.*

In the rest of the paper we discuss this framework in more detail. In Section 2 we describe the problem in more detail. Section 3 discusses the advantages of expressing design as a logic meta-program of implementation. In Section 4 we then introduce the logic meta-programming language we implemented and that we use to relate design and implementation. In Section 5 we show the *Declarative Framework*, a library of logic rules that implement relations between existing design techniques (UML class diagrams and design patterns). In Section 6 we then introduce the synchronisation framework, and show the kinds of synchronisation it supports. Section 7 describes the experiments we did on two separate case studies (the HotDraw framework and a real-world application) to show the viability of the approach in general and of our implementation in particular. Based on the results from the experiments, Section 8 discusses our solution. Section 9 concludes the paper.

2 The Gap Between Design and Implementation

In traditional software engineering literature, implementation is typically viewed as a concretization of design [Bud94, GR95, Som96]. This implies a very general, unidirectional mapping from design to implementation. General forward engineering techniques do not even bother with making this mapping between design and implementation explicit. This implicitness leads to serious problems when developing object-oriented systems, as shown by the following well-known problems:

Drift and erosion. *Drift* occurs where implementation and design evolve in different directions because they are not explicitly related. *Erosion* is the process where the initial design is breached more and more in the implementation, because the design becomes less and less relevant as the implementation changes to accommodate new requirements [PW92]. This

term describes the problem very well: over time the artefacts from the original phases erode more and more under the constant pressure of the ever changing implementation.

Documentation problems. Severe problems occur when one documents a system and has to keep this documentation up-to-date. Documentation of a system is not only needed when maintaining a system, but also when reusing (part of) it or when adding new requirements. For all these activities, documentation is needed so that the system can be fully understood before making changes to it.

Supporting iterative development. *Iterative development* is targeted more towards the development of a system built for new domains and with changing user requirements. The strong point of iterative development is that it integrates top-down development (typically done in the design phase) with bottom-up development (typically encountered when implementing the design). In each pass, the implementation learns from the design, and the design learns from information gathered in the implementation phase. This integration of top-down and bottom-up development makes iterative development much more reactive towards changing requirements and reuse. However, this flexibility comes at a cost: *synchronisation*. Properly supporting iterative development is impossible if the design phase and the implementation phase (through which is continuously cycled) have to be synchronised manually. This is not a shortcoming of incremental development alone; it just shows how crucial it is to be able to synchronise design and implementation.

The fundamental problem underlying the problems sketched above is that the is *relation between the design and the implementation is not explicitly recorded*. Because design and implementation are unrelated, they can be modified independently of each other, and a modification of either one does not leave any trace in the other. As a result, synchronising such two loosely coupled entities is at best difficult and ad-hoc, and most of the time impossible. This discrepancy results in a practical development process where analysis and design are used for the initial implementation, but evolution is applied to the implementation alone [DDVMW00].

3 Design as a Logic Meta-Program over Implementation

In this paper we relate design and implementation in such a way that we support the synchronisation of changes to design and implementation. Before we look at our solution, however, we want to stress that in our view design defines a complete range of abstractions from implementation. This spectrum of abstractions ranges from more local and detailed design (such as programming styles [Jon87, LH89, Bec97]) to high-level abstractions that provide global views of the implementation (as in high-level design or software architectures [PW92, GS93, BJ94, BMR^{+}96, SG96]). Hence the relation between design and implementation will not be as straightforward as what is assumed in existing tools (like for example *TogetherJ* or *Rational Rose for Java*).

The cornerstone of our solution is that design is expressed as a logic meta-program over implementation. Expressing design as a logic meta-program over implementation has several advantages:

1. the relation between design and implementation is made explicit, since the design is expressed in terms of the implementation;

2. we benefit from the open character of a logic programming language, which allows us to build a system where rules can easily be added to implement specific behaviour, and where logic repositories are used to group and nest rules;

3. the inherent declarative nature of logic programming is very well suited to express design notations, since these are typically also declarative in nature;

4. since all design notations we support are expressed in the same medium (as logic meta-programs), they can be expressed in terms of each other. For example, the structure of design

patterns [GHJV94] can be described using UML class diagrams [RJB99], taking *best practice patterns* [Bec97] or other programming conventions into account. And finally

5. logic programs express relations between their variables, in a mathematical sense. This property is also referred to as the *multi-way property* of logic programming languages. Concretely this means that the same logic program can be used in many different ways depending on the information that is passed to it.

4 SOUL

We have seen why we would like to express design as a logic meta-program over implementation. In this section we look at the programming language that allows us to do this, called the *Smalltalk Open Unification Language* (*SOUL*). SOUL is a logic programming language (analogous to Prolog [CM81, SS88]) that is implemented in, and lives in symbiosis with, the object-oriented programming language *Smalltalk*. SOUL allows users to perform logic queries over Smalltalk source code, without the need of representing this source code explicitly in the logic repository. This is accomplished by incorporating a special term called the `smalltalk term`, and a special predicate called the *generate predicate*. Next two sections describe these additions.

4.1 The Smalltalk Term

The SOUL language construct that enables symbiosis is the `smalltalk term` that allows us to use Smalltalk objects as logic constants. So where Prolog uses *symbols* and *numbers* as constants, SOUL uses any Smalltalk object. Take for example the following query:

Query foo([Array])

The argument of *foo* (the term with the square brackets) is a `smalltalk term`. Between the square brackets we find the description of the term, in this example *Array*. *Array* is an object, namely the object that represents the class *Array*.

What we have not discussed yet is how the reified objects can be specified or, in other words, what can occur between the square brackets. The `smalltalk term` allows any Smalltalk expression (like the very simple expression *Array* that evaluates to the object Array). Moreover, these expressions allow logic variables as receivers of messages. We illustrate this with a query that binds a logic variable *?c* to a `smalltalk term` *[Array]*, and then uses a `smalltalk term` containing a piece of code that evaluates to a boolean. This piece of code sends the message *selectors* to the binding of *?c*, which returns a collection object that contains all the names of methods. This collection object is then sent the message *isEmpty* that returns the object *true* when the collection is empty and *false* otherwise:

Query equals(?c, [Array]),
[?c selectors isEmpty]

Interpreting this query will succeed if the class *Array* has methods, and will fail otherwise. Note that for this query to succeed, we did not need to write any logic fact expressing information about *Array*: this was all done by using the Smalltalk system.

4.2 The Generate Predicate

The `smalltalk term` wraps Smalltalk expressions, and allows us to evaluate Smalltalk expressions during logic interpretation, wrapping the resulting object. This wrapping was illustrated in the

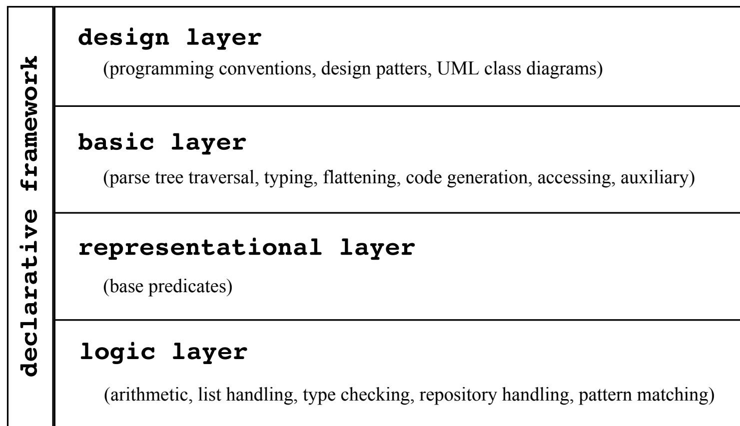

Figure 2: The declarative framework

previous section, where the query had one result, a wrapped Smalltalk collection containing the selectors of class *Array*. However, sometimes we want to get *separate* logic results for each selector. Such functionality is offered by the *generate predicate*, which generates a set of solutions (described by a `smalltalk term`) for a variable. The first argument of the *generate predicate* specifies a *logic variable* to bind the results to. The second argument is a `smalltalk term` that describes a *stream of solutions*. Each of these solutions is bound, one by one, to the first argument. As an example, we implement a *simpleSelector* rule that expresses the relation between classes and selectors (names of methods) of that class. It first expresses that the *?c* is a class, and then uses the *generate predicate* to ask this class for its selectors and bind these to the *?selector* argument. Asking the class for its selectors is done by the `smalltalk term`, that sends the message *selectors* to the class bound to *?c*, and then sends *asStream* to return a stream of solutions:

```
Rule simpleSelector(?c, ?selector) if
        class(?c),
        generate(?selector, [?c selectors asStream])
```

When the *generate predicate* is evaluated, it results in n solutions for the variable *?selectors*, where n is the number of elements in the stream. For example, the query to ask the selectors of class *Array* now yields 15 results. Each result is a binding for the variable *selector*, containing the name of a method of *Array*.

5 The Declarative Framework

The declarative framework is a layered rulebase of rules expressing design as a logic meta-program over implementation. It is depicted in Figure 2. Rules in one layer have access to all the rules from lower layers. Each layer contains groups of rules with similar or related functionality:

- The *logic layer*: this layer contains the rules that add core logic-programming functionality, such as *list handling*, *arithmetic*, *program control*, *repository handling* and *SOUL meta-programming*.
- The *representational layer*: this layer reifies the base-language's concepts, such as *classes*, *methods*, *instance variables* and *inheritance*.

- The *basic layer*: this layer adds a lot of auxiliary rules that facilitate reasoning about implementation. Since the *representational layer* only provides the most primitive information, this layer is necessary to interact at a reasonable level of abstraction with the logic meta-programming language.

- The *design layer*: this layer groups all rules that express particular design notations. In the next chapter we describe some design notations that we have expressed in our experiments, namely *programming conventions*, *design patterns* and *UML class diagrams*.

We now look at the layers in more detail, and give SOUL examples to demonstrate the expressiveness of the language. Since this paper focuses on synchronising design and implementation we omit a detailed description of the logic layer.

5.1 The Representational Layer

The representation layer is responsible for reifying the concepts of the base language that we want to reason about in the rest of the declarative framework. We reify four concepts from the base language: *class*, *method*, *instance variable* and *inheritance*. All the other rules in the declarative framework use these concepts to reason about the implementation. In this section we show what the rules for reifying classes and methods look like in SOUL. Since the rules for instance variables and inheritance are similar, we do not explicitly show these.

5.1.1 The *Class* Rule

We start with the implementation of the *class* rule. This rule allows us to check if its argument is a class, or to generate all classes. Note that the `smalltalk term` uses the class *SOULExplicitMLI* as a facade to facilitate and centralise the calls to the Smalltalk system[1].

Rule class(?c) **if**
  generate(?c, [SOULExplicitMLI current allClasses]),

This rule uses the *generate predicate* to generate all classes in the system. One by one it binds these values as the result of the *?c* variable. For example we can then perform a query asking for all the classes in the system:

Query class(?c)

However, it also solves the following query that asks whether the argument *Array* is a class:

Query class([Array])

Note that we actually pass an *Array* class here (and not some description), using the fact that classes are first-class objects in Smalltalk and that we can represent and use these objects using `smalltalk terms`.

[1]The implementation of the facade *SOULExplicitMLI* actually uses a singleton design pattern [GHJV94], which explains the *current* message that is sent to the *SOULExplicitMLI* class to retrieve the actual facade instance used. This instance is then asked for all the classes in Smalltalk using the *allClasses* message.

5.1.2 The *Method* Rule

Analogous to the *class* rule, we define a *method* rule that allows us to represent methods of classes in a logic form. We represent a Smalltalk method as a functor with five arguments: the class, its name, the names of the arguments, the names of the temporary variables and the statements. The mapping we use to represent Smalltalk programs in the declarative framework is fairly straightforward, and follows the Smalltalk parse tree structure. We do not show the actual mapping in detail, as this falls outside the scope of this paper. Interested readers can find the details in [Wuy01].

```
Rule method(?c, ?m) if
        class(?c),
        generate(?method, [SOULExplicitMLI current allMethods]),
        equals(?m, ?method).
```

Once this rule is added (together with the *class* rule defined above), we can reason about classes and methods in the object-oriented system. This allows us to perform queries such as asking whether a specific method is indeed a method of some class, finding all the methods of a class, and so on. This will become clear in examples of the other layers.

5.2 The Basic Layer

The *logic layer* and the *representational layer* provide all the basic mechanisms to reason about Smalltalk code. However, the level of abstraction is not very high, and for almost every query we should have to write lots of logic code. Therefore we factored out a lot of functionality and created the *basic layer*. This layer adds a lot of auxiliary rules that facilitate the reasoning about implementation, and raises the level of abstraction significantly. Describing the implementation of all of these rules falls outside the scope of the paper (there are currently over 140 rules in this layer). Therefore we summarize the rules in groups, and then give some examples on how to use them.

- *Parse tree traversal*: a lot of rules have to traverse the parse tree of a method to search for certain variables or message sends. This group implements parse tree traversal, and some commonly used traversals (looking for receivers or messages sent, for example).
- *Typing*: Smalltalk is dynamically typed. Therefore we added some rules that analyse the source code in order to infer possible types for variables.
- *Flattening*: the basic rules representing classes and methods are *incremental*, meaning that the information about a class reflects only what that class implements and not what it inherits. The rules in this group allow us to reason about classes in their flattened versions.
- *Code generation*: since smalltalk rules can contain any Smalltalk code, this code can use the standard Smalltalk meta facilities to generate or remove code. The rules in this group use this feature to generate code from logic descriptions of methods or from quoted strings.
- *Auxiliary*: there are also a number of auxiliary methods (such as *rootClass*, *hierarchy*, *understands*, *abstractClass*) that implement various frequently used rules.

To see what can be done with the rules in this layer, we start by writing a rule *varInterface* that describes the interface for an instance variable as all messages sent to this instance variable. It uses a *findall* to collect all sends to the variable in a list. The sends to the variable are found by the *isSendTo* rule, one of the parse tree traversal rules. The interface is this set of methods, but without any duplicates:

```
Rule varInterface(?class, ?var, ?interface) if
        instvar(?class, ?var),
        findall(         ?varSend,
                         isSendTo(?class, _, ?var, ?varSend),
                         ?varSendsList),
        noDups(?varSendsList, ?interface).
```

Using the *varInterface* rule we can then write a rule *interfaceDifferences* to find all sends to an instance variable that are not understood by some class. This allows us to check whether some class we see as a possible type for the instance variable could indeed be used as such. The implementation simply gets the interface of the instance variable (using the *varInterface* rule), and extracts all selectors from it that are *not* understood by the type (that has to be a class):

```
Rule interfaceDifferences(?class, ?var, ?varType, ?missingSelectors) if
        class(?varType),
        varInterface(?class, ?var, ?interface),
        findall(         ?missingSelector,
                         and( member(?newSelector, ?interface),
                              not(understands(?varType, ?missingSelector))),
                         ?missingSelectors).
```

For example, using this rule we can check whether *Number* is a possible class for the instance variable *x* of class *Point*:

```
Query     interfaceDifferences([Point], [#x], [Number], ?missing)
```

As could be expected, the query succeeds and returns as only possible value for the *?missing* variable the empty list. This means that *Number* is a possible class for the instance variable *x*, at least by looking at the messages sent to *x*.

The *interfaceDifferences* rule can only be used in a more constructive way than to report on the missing methods. We can for example also use it to generate skeleton methods for every method that *interfaceDifferences* reports as missing. This situation arises when implementing a core class before its auxiliary classes. Then one ends up with a class which implementation is nearly finished, and then has to implement all the cooperating classes. The interfaces of these cooperating classes, however, are already defined by the core class. Using the *interfacceDifferences* rule we can support this scenario by extracting all the needed methods and generating template methods on the cooperating classes. The following rule implements such support. Note that the *notYetImplementedSource* gives the template source code for a given selector).

```
Rule adjustClass(?class, ?var, ?type) if
        interfaceDifferences(?class, ?var, ?type, ?missingSelectors),
        forall(member(?sel, ?missingSelectors),
              and( notYetImplementedSource(?sel, ?code),
                   generateMethod(?type, ?code))).
```

5.3 The Design Layer

The top layer, that builds on all the other ones, is the *design layer*. It contains rules that express *programming conventions*, *design pattern structures* and *UML class diagrams* in terms of the implementation. The following sections give examples from the design pattern and the UML rules.

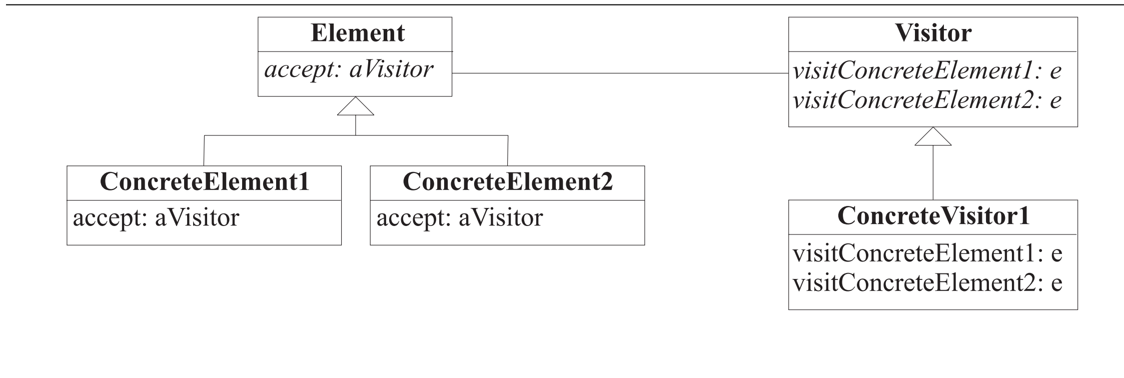

Figure 3: Visitor Design Pattern Structure

5.3.1 Design Pattern Structures

We expressed structures as described by design patterns [GHJV94]. In general, a design pattern is detectable if its template solution is both distinctive and unambiguous [Bro96]. The design pattern structures we have expressed in this layer are the *Composite Pattern*, *Visitor*, *Abstract Factory*, *Factory Method*, *Singleton* and *Bridge*. In this section we give the implementation of the *visitor* design pattern structure.

The general idea of the *Visitor* design pattern is to separate the *structure* of elements from the *operations* that can be applied on these elements. This separation makes it easier to add new operations, because the classes of the object structure do not have to be changed. The typical example of the *Visitor* pattern is to separate parse trees from the operations that are typically performed on these parse trees (such as generating code, pretty printing or optimizations). The general structure of the *Visitor* pattern is depicted in Figure 3.

The rule describing the structure of the *Visitor* pattern is fairly straightforward. It expresses first of all that the *visitor* is a class, and that it implements the *visit* methods (that have the name *visitSelector*). In the same way, *element* is a class too, and implements methods called *accept* with a body *acceptBody*. The arguments passed to this method are given by *acceptArgs*. The body is responsible for calling the passed visitor *v* with the actual visit operation *visitSelector* and passing along the arguments *visitArgs*. One of the arguments has to be the receiver (denoted by *self* in Smalltalk), and the passed visitor *v* actually has to be an argument of the accept method:

```
Rule visitor(?visitor, ?element, ?accept, ?visitSelector) if
    class(?visitor),
    classImplements(?visitor, ?visitSelector),
    class(?element),
    classImplementsMethodNamed(?element, ?accept, ?acceptBody),
    methodArguments(?acceptBody, ?acceptArgs),
    methodStatements(  ?acceptBody,
                       <return(send(?v, ?visitSelector, ?visitArgs))>),
    member(variable([#self]), ?visitArgs),
    member(?v, ?acceptArgs).
```

5.4 UML Class Diagrams

The *UML class diagram* rules express the basic concepts of UML class diagrams [BRJ97, RJB99]: *classifiers* (with *operations* and *attributes*) and the *generalization* and *association* relationships.

In this case we take the most complicated one, namely *mapUMLAssociation*. This rule is used to map UML associations against the source code. Because Smalltalk is dynamically typed, extracting and checking collaborations between classes is hard. However, we can use the *typing rules* to extract possible associations. The core of mapping the UML association relation to the implementation is the *associationRelation* rule, that types the instance variables of the left class (using the *instvarTypes* and *stripHierarchyClasses* rules) and uses that information to see if there is an association with the right class. This is done by taking the instance variables of the left class and, for each of them, determining their type. If the type is not a Smalltalk collection, then the multiplicity is set to 1. If the type is found to be some Smalltalk collection class, then the multiplicity is set to *many*, and the type of the elements contained in the collection is determined (with the *collectionElementType* rule). For each possible type we then construct a *role* functor with the extracted information (type and multiplicity). Since the *?allRoles* then contains possible nested lists, we flatten the results before returning them:

```
Rule associationRelation(?leftClass, ?instvar, ?leftRoles) if
        instVarTypes(?leftClass, ?instvar, ?typeList),
        stripHierarchyClasses(?typeList, ?possibleTypes),
        findall( ?roles,
                and(member(?possibleType, ?possibleTypes),
                     or( and(containerType(?possibleType),
                               collectionElementType(?leftClass, ?instvar, ?types),
                               stripHierarchyClasses(?types, ?strippedTypes),
                               findall( role(?instvar, multiplicity([#many]), type(?possibleType, ?type)),
                                        member(?type, ?strippedTypes),
                                        ?roles)),
                          and(not(containerType(?possibleType)),
                               equals(?roles, <role(?instvar, multiplicity([1]), type(?possibleType))>)))),
                ?allRoles),
        flatten(?allRoles, ?leftRoles).
```

6 The Synchronisation Framework

We now have discussed the key part of the synchronisation framework (namely that design is expressed as a logic meta-program of implementation). We have also discussed the declarative framework, shown examples on what the relation between design and implementation looks like and how it can be used. This forms the core of the synchronisation framework as depicted in Figure 1 in Section 1. The integration with the development environment is done with two monitors, that notify us of changes in respectively design and implementation. Hence any change to design or implementation can be intercepted, and the synchronisation can start.

We already indicated that the synchronisation framework supports a whole spectrum of synchronisation. This spectrum can be characterized by different axes [Wuy01], of which we will now see the two most important ones: *trigger time* and *direction of synchronisation* (discussing the other axes falls outside the scope of this paper). We then introduce some of the applications that were built using the synchronisation framework.

6.1 Direction of Synchronisation

Although there are two partners to be synchronised (design and implementation), synchronisation does not necessarily works in both directions. When only one partner can be derived from the other, we have a *unidirectional* synchronisation. With a *bidirectional* approach, design can be derived from implementation and vice versa. This classification has a strong impact on the results that can be expected from the synchronisation: a unidirectional system can only be used to generate one of the two participants from the other, or to do a limited conformance check. A

bidirectional system can be used both for conformance checking and for generating one participant from the other and vice versa.

The synchronisation framework we propose supports both unidirectional and bidirectional synchronisation:

- When both design and implementation are given, we can run queries to check for differences between both. Suppose for example that we have an implementation and that we have documentation stating that is implemented according to a visitor design pattern. Then we can use the *visitor* rule we described in Section 5.3.1 to find the differences between design and implementation.
- When only the design is given, we can generate implementation skeletons. Of course, these skeletons will then need to be completed manually. How much can be generated depends on the design mapping rules.
- When only the design is given, we can extract the design from the implementation. Suppose for example that we have an implementation for which we have no design information. Then we can for example run a query to check whether there is a visitor design pattern in this implementation.

The most interesting feature of our approach is that the same mappings can be used for both directions. This is possible because we express design as a logic meta-program of implementation. The logic programming language takes care of the rest.

6.2 Trigger Time

The synchronisation can be triggered *directly* after every single change, or *delayed*, after several changes were made. Because the monitors can intercept changes to design or implementation, we can support both models. When the monitors intercept every change directly, we can invoke the synchronisation engine to do for example a conformance check. The results of this check can then be used to for example notify the developer that a certain implementation change was not allowed or that a certain design change is not consistent with the implementation. When the notifiers are only set to react on certain changes, or not at all, we have support for delayed synchronisation. For example, the synchronisation can be invoked manually after any number of changes.

6.3 Applications using the Framework

On top of the Synchronisation Framework we have built some applications to show the possibilities of the framework:

Find Tool. This tool is an application that allows you to find classes and methods according to certain criteria that can be selected from drop-down boxes (such as is used in search facilities offered in graphical operating systems). Using this application, it is easy to look for all classes that belong to a certain hierarchy, that implement a method which name begins with an 'a' and that implement at least one abstract method.

Style Checker. This application acts as a monitor for the quality of methods in the system. Therefore it fires user-definable queries whenever a method is changed in the system. These queries express criteria methods should comply to. Failures to adhere to these criteria results in a warning being written to the log application. The entries in the log application can then be double-clicked to open a browser on the method that causes the warning. Entries in the to-do log are overridden when the same method is changed (and thus always contain the results of the latest version of the method), or can be removed manually. The developer is thus not hindered: violations only result in logs to be reviewed afterwards.

UML Tool. A third tool we implemented is a UML Tool that allows one to extract, view and manipulate UML Class Diagrams from Smalltalk code. Moreover, it can also generate code skeletons, given a diagram. We could then go one step further and have used the notification system to act on changes in system or design. For example, we can then make a UML tool that enforces that every change in a design diagram should comply to the implementation. For example, when an operation is added to a classifier, the implementation can be checked to make sure that the class corresponding to the classifier indeed implements such method. If not, this can be logged or template code might be generated.

7 Experimental Validation

The previous sections introduced the Synchronisation Framework, and showed how it supports synchronisation. In this section we now describe some experiments we did in order to validate whether this framework is indeed capable of synchronising changes between design and implementation in practice. We conducted two series of experiments:

HotDraw. We performed experiments on the well-known HotDraw framework. We were interested in showing the different possibilities of the synchronisation framework, and in the practical applicability of the implementation of the framework.

Real-World Case. We also used the synchronisation framework on a large-scale industrial application to assess the usability and scalability in that context.

Because of space constraints we can only give an overview of the experiments we performed, but more detailed information can be found in [Wuy01].

7.1 The HotDraw Experiments

Through an extensive case study of the *HotDraw* framework, we showed how the synchronisation framework can be used to:

- extract design (class diagrams, design pattern structures) from the implementation;
- generate implementation from the design;
- do a conformance check between design and implementation, showing discrepancies between both;
- guide implementation by checking implementation changes against design information, and possibly reacting on these changes by generating code.

We also showed how we can use the synchronisation framework to make some undocumented and hard to find relations between the *DrawingEditor*, *Figure* and *Tool* classes explicit. These relations are hardcoded in a number of methods on these three classes, and use several naming conventions and low-level dependencies. Using SOUL we made these conventions explicit and used them to guide development. This shows the practical usability of synchronisation, even for a mature and refined framework that formed the basis for several design patterns.

7.2 The MediaGeniX Experiments

The real-world tests were performed at MediaGeniX, a company that develops tailor-made broadcast management systems for television stations. For our tests we worked on the *Media Management* module of *Whats'On*, that handles everything that has to do with the actual management of the media used for broadcasting, such as tapes. This module has recently been rewritten, and consists of 441 classes. Since the *Media Management* module is one of the newer parts of

Whats'On, it is one of the first to use the *MediaGeniX Application Framework* (MAF). The MAF is MediaGeniX' framework for building applications. We performed two sets of experiments. The first was to synchronise the existing UML diagrams from the *Media Management* module with the implementation. The second was to make explicit the rules that MAF applications should comply with, and to check existing applications for conformance with these rules.

The MediaGeniX experiments shows that the synchronisation framework can be used in a practical setting to synchronise design and implementation. In a limited period of time we successfully applied the synchronisation framework to express and synchronise design information with an implementation. More specifically, we did a conformance check of existing UML diagrams with the released implementation. We found some discrepancies between the two, most notable some classes and relations in the UML diagram that did not exist in the implementation. Also, we were able to complement the UML diagrams with information we extracted, most notably role names for associations. We also checked the evolution of the implementation with respect to this UML diagram. Besides these experiments with UML diagrams, we also made a set of programming conventions explicit, and used this to find violations against these programming conventions in the implementation. The results of these checks where a number of clear errors in some parts of the implementation that do not follow the programming conventions.

Overall, the experiments on *HotDraw* and *Whats'On* showed that the rules in the declarative framework, although lightweight, can be used to successfully express the design used in a particular context and that the synchronisation framework successfully synchronises design and implementation.

8 Discussion

In this section we discuss some of the results that came up during the experiments.

Power of LP. The reasoning power of SOUL is necessary to express several design relations. For example, rules for expressing design patterns or UML class diagrams need the full power offered by a logic programming language. ONe of the reasons is transitive closures of methods sends need to be taken into account to express certain programming conventions. This is hard to express in approaches that do not support recursion (such as *SmallLint* [RBJO96], a regular-expression based tool). However, to express some of the other programming conventions, no recursion is necessary and hence less powerful but faster reasoning engines could be used (such as SmallLint).

Extensible rule system. The declarative framework is the key mechanism in being able to adapt quickly to different implementations. For example, being able to complement the general rules with two *MediaGeniX* -specific rules meant we could reuse the declarative framework. Actually, two features are necessary: a composition mechanism of repositories, and a mechanism that clauses can easily use and override other clauses.

Performance. Not performance but extensibility was our main concern when implementing SOUL. However, based on the experiments we feel that we can safely say that even using the current, non optimized implementation of SOUL, querying the source code using a logic meta-programming language is feasible. While larger queries (with complicated mappings to the source code, such as in our paper expressing software architectures [MWD99]), queries might take longer than an hour, most queries take on the order of minutes (on regular desktop machines). This is too slow to be truly interactive, but the speed of commercial Prolog implementations makes these kinds of queries possible.

Scalability. The key point in making the approach efficient on large systems is reduction of scope. This can be done by making use of the programming conventions, and by pre-filtering irrelevant information using a coarse grained (but efficient and inexpensive) approach, and then finecombing these results with the more expensive full logic programming approach;

9 Conclusion

The problem addressed by this paper is that the relation between design and implementation should be made explicit so that changes to design and implementation can be synchronised. As solution we introduce the synchronisation framework, that is based on the following three core concepts:

1. make the relation between design and implementation explicit by expressing design as a logic meta-program of implementation;
2. integrate the logic-meta programming language in the environment to capture changes of design and implementation;
3. use the logic meta-programming language to find differences between design and implementation, and define actions.

To show that this solution is not only a conceptual solution, but can also be used in practice, we have implemented the *synchronisation framework*. Therefore we first of all implemented a logic meta-programming language that exploits its symbiosis with the base language to reason directly over the implementation of the base programs. Then we wrote logic meta-programs to express design in terms of implementation, and layered these programs in the declarative framework. Everything was then further integrated with the development environment by adding design and implementation monitors.

To show its usability and scalability in practice, the synchronisation framework was applied to two different case studies. This experimentally shown on a smaller case study (the HotDraw drawing editor framework) that the synchronisation framework indeed supports all different characterizations of synchronisation. On the same case study, we also found that, even for the well-known and well-documented framework HotDraw is, there is need for synchronisation of design and implementation and the synchronisation framework can do so. Besides the experiments on HotDraw we also did experiments on a large industrial framework. Here we did conformance checks of UML diagrams against the implementation (complementing the diagrams with extracted information and detecting differences between the UML diagrams and the implementation). We also expressed programming conventions and found several violations in the implementation that needed to be fixed. These experiments strengthened our claim that the synchronisation framework is usable in practice, and showed that it is scalable.

References

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

[BJ94] K. Beck and R. Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, volume 821 of *LNCS*, pp. 139–149. Springer-Verlag, July 1994.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, 1996.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Method Language User Guide*. Addison-Wesley, 1997.

[Bro96] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, 1996. TR-96-07.

[Bud94] D. Budgen. *Software Design*. Addison-Wesley, 1994.

[CM81] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.

[DDVMW00] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of the international symposium on Software Architectures and Component Technology 2000.*, 2000.

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

[GR95] A. Goldberg and K. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management*. Addisson-Wesley, 1995.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering, V. Ambriola and G. Tortora (eds.)*, volume I. World Scientific Publishing, 1993.

[Jon87] W. C. Jones. *Modula2: Problem Solving and Programming with Style*. Harper & Row, 1987.

[LH89] K. Lieberherr and I. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pp. 38–48, September 1989.

[MWD99] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pp. 33–45, June 1999.

[PW92] D. Perry and A. Wolf. Foundations for the study of software architectures. *SIGSOFT Software Engineering Notes*, 17:40–52, October 1992.

[RBJO96] D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

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

[Som96] I. Sommerville. *Software Engineering*. Addison-Wesley, 1996.

[SS88] L. Sterling and E. Shapiro. *The art of Prolog*. The MIT Press, Cambridge, 1988.

[Wuy01] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.