Knowledge-based Techniques to Support Reuse in Vertical Markets

Ellen Van Paesschen [*]
Ellen.Van.Paesschen@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium

April 17, 2001

Abstract

Vertical markets supported by framework technology experience a range of difficulties when performing domain analysis and reusing frameworks, especially during the transition between these two phases. We believe that these difficulties are caused by the implicit nature of the natural relationship between domain models (and the corresponding delta-analyses) and the framework instances. We aim to provide a mechanism for the construction of a bidirectional, concrete, active link between domain models and framework code by representing the results of delta-analyses as "knowledge", and by supporting the coupling between the explicitated deltas and the knowledge at implementation level, with A.I. reasoning techniques. This approach implies the realization of a "one-to-one" cycle between domain analysis and framework instances, resulting in an optimization of the reuse of frameworks. Starting from the existing co-evolution research for design and implementation at our laboratory, this cycle will ensure that analysis, design and implementation evolve together whenever changes occur to one or more of them.

1 Introduction

1.1 Co-evolution of Design and Implementation

In recent years, several members of our laboratory have conducted a great deal of research on the subject of co-evolution [1] of design and implementation ([5],

[*]Author financed with a doctoral grant from the Institute for Science and Technology (IWT), Flanders

[1]Co-evolution is inspired by biology, where it describes the process whereby organisms interact and mutually influence genetic characteristics. In the case of software, it describes the way different phases in the life cycle of some system (initially design and implementation) evolve and are causally connected in some way. Not to be confused with the concept "co-evolution" in the context of genetic programming.

[11], [4], [6], [10]). This co-evolution consists of the realization of an active link between design and implementation in such a way that they can evolve together whenever changes occur to one of them. At this moment, design is being considered as declarative knowledge about the implementation. This declarative knowledge is coded into logic rules, as in prolog, and into knowledge bases. Well-known reasoning algorithms (backward and forward chainers) are being used to couple the knowledge (design) to the implementation in an active manner. Whenever changes occur in design or implementation, the necessary inference algorithms are activated in order to propagate the changes to the other model.

As a representative example of the ongoing co-evolution research at our laboratory, we will focus on the the Smalltalk Open Unification Language (SOUL) [11]. Wuyts [11] developed this logic meta-programming language for his PhD thesis to express design as a logic meta-program over implementation and to synchronize design and implementation. SOUL is a logic programming language (analogous to prolog) 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 done with the `smalltalk term`, a special construct that allows to invoke Smalltalk code during the logic interpretation process. Using the `smalltalk term`, concepts from the base language can easily be reified in SOUL.

This logic meta-programming language makes it possible to express design as a logic meta-program over implementation in a structured manner, resulting in a declarative framework. An example of using this technique is to describe design patterns as a collection of logic rules and facts. To illustrate this, we include an example describing the design pattern structure rule for the Visitor pattern:

```
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).
```

This rule describing the structure of the Visitor design pattern expresses that the `visitor` is a class, implementing the visit methods (named `visitSelector`). Analogously, `element` is a class that 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 for 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.

We can apply the Visitor predicate described above in a mind boggling way: since SOUL is implemented in Smalltalk and uses a Visitor pattern itself, the Visitor predicate that is written in SOUL, can be applied to SOUL's own Visitor pattern. SOUL uses this Visitor pattern to enumerate its parse tree, therefore, the Visitor predicate can be used to find all the non-abstract parse tree elements of the SOUL parse tree that do not comply to the Visitor pattern. All subclasses of class `SOULParseTreeElement` that are not abstract are selected, from which all classes that do not comply to the Visitor rule are found:

```
Rule soulParsetreeVisitor(?node) if
hierarchy([SOULParseTreeElement], ?node),
not(abstractClass(?node)),
not(visitor(?visitor, ?node, [#doNode:], ?callbackMsg))
```

The last line in this rule gives the name of the visit-method used by the Visitor to visit the nodes. This Smalltalk Symbol is called `doNode`. The results of this query contain the methods that do not comply to the SOUL Visitor design pattern, and that might need to be changed. If the query fails, then all the classes and methods comply to the Visitor design pattern. For more details on SOUL, we refer to [11].

Naturally, SOUL supports other design notations besides design pattern structure rules, in fact, since SOUL is analogous to prolog and prolog is a Turing-computable formalism, everything that is computable can be expressed in SOUL: "the computability is the limit".

Currently, several members of our laboratory are improving different aspects of the corresponding co-evolution technology. From our point of view, one of the most interesting future work directions of SOUL consists of the development of full co-evolution support, i.e. to take into account other phases of the development cycle besides design and implementation.

1.2 Adding Analysis to the Existing Co-evolution of Design and Implementation

Inspired by the promising results of this co-evolution research, we propose to integrate the analysis phase in the existing co-evolution of design and implementation. To implement this full co-evolution support we have chosen the domain of vertical markets supported by framework technology.

For each customer, vertical market companies adapt the same framework, until it meets the customer's requirements, and then instantiate this framework. In this way, companies in a vertical market tackle the duality of *tailor-made* and *off-the-shelf* software [2]. In this context, the classical analysis phase is combined with a *domain analysis* with the purpose of constructing a *domain model.* To define which functionalities a new customer demands from his future system, a *delta analysis* is performed on this domain model, resulting in what is known as the *deltas*, describing the differences in functionality between an instance of the current framework and the system that is desired by the customer.

Although there exists a “natural” relationship between the deltas of the delta analysis and the future framework instance, this relationship is merely implicitly present in the minds of analysts and framework engineers. The result of this implicity is that frameworks are developed and reused in a very “handcrafted” way, based on delta analyses ([2]).

The absence of a concrete technical link between deltas and framework implies a range of problems that are experienced during framework development and reuse. First, it becomes difficult to map functionalities onto the implementation of a framework and vice versa. This implies that analysts do not longer know what the exact possibilities of the framework are. Second, it is hard to map variations in functionality onto the variabilities of the framework and vice versa. Third, the missing link between deltas and framework implementation causes the classical transition problem from analysis to implementation to be increased. Finally, it becomes difficult to determine the reusability and the corresponding reuse scenarios of a framework. This last difficulty is increased by the frequent absence of suitable framework documentation.

Our position is that, in order to make the current framework development and reuse methodologies more structured and thus to avoid the above problems, the currently implicit relationship between deltas and framework code should be explicitated into a technological, bidirectional, concrete, active link. Our approach will apply existing techniques from the domain of Artificial Intelligence to realize a coupling between the (domain) analysis level and the framework implementation level.

Figure 1 illustrates this approach. The active link between domain model and framework will be some kind of expert system that enables to:

1. Plan a reuse scenario for developers driven by the domain model and corresponding delta-analysis

2. Distill knowledge from the iterative adaptations and extensions of the framework core and to integrate this knowledge in the domain model to use it during future delta analyses

In the first direction, the expert system should couple domain model knowledge and the framework implementation. In the second direction from framework to domain model, this expert system will translate framework implementation changes in terms of the corresponding domain model so that the analysts - at all time - know about the possibilities of the framework.

Although in recent years there was conducted a lot of research on efficient framework reuse - e.g. design patterns -, it appears that the topic of reuse techniques manipulated delta analyses has mostly been neglected. Also the influence of these reuse techniques on the delta analysis has not been investigated sufficiently. Nevertheless, it is exactly the coupling between deltas and framework implementation that is crucial for the successfulness of vertical markets companies that apply framework technology.

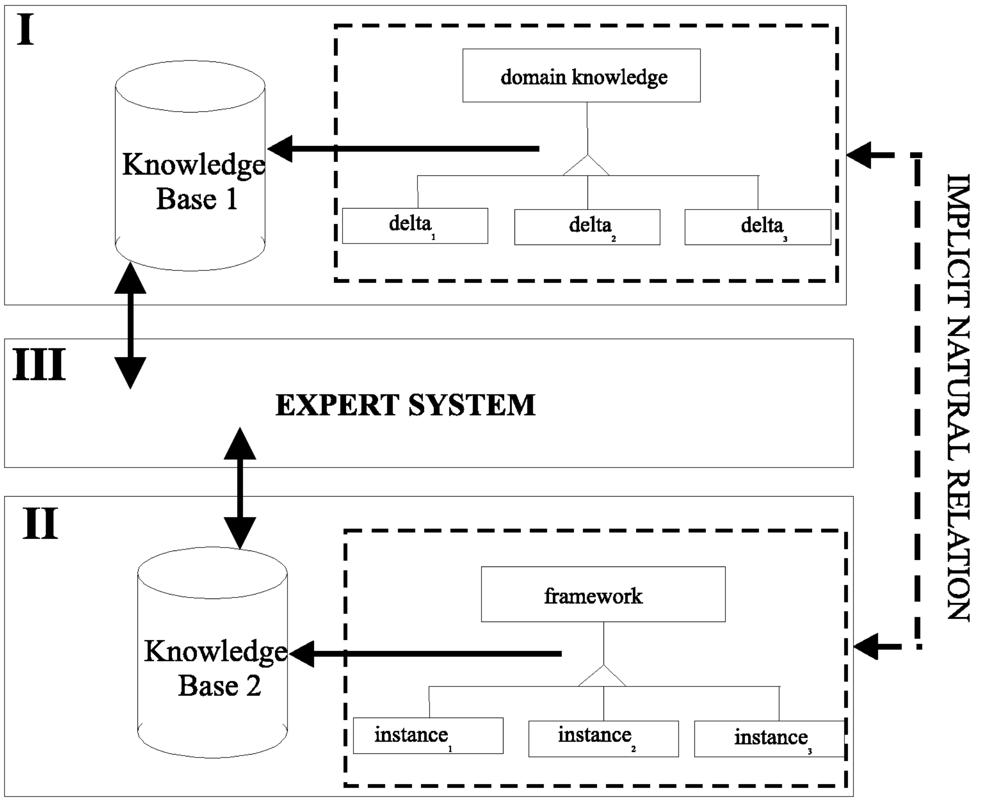

Figure 1: Coupling the analysis and the framework level

A more detailed description of our approach and the experiences at this early stage of our research are described in the following sections.

2 Constructing the Link between Domain Models and Frameworks

We have divided the construction of an active link between domain models and frameworks into three parts: the identification a suitable knowledge representation for domain models and the corresponding deltas, the identification of a logic representation for framework design and, finally, the construction of an expert system for linking these domain and framework knowledge. In the following sections, all three tasks will be discussed.

2.1 A Knowledge Representation for Domain and Delta Analyses

To represent domain knowledge and deltas we want to use existing knowledge representations from A.I. like predicate logic and/or frame-based techniques [9]. The exact representation has not been decided yet. At the moment several representations are being evaluated depending on the following aspects:

- The representation requirements of existing object-oriented analysis techniques and existing domain analysis techniques [3]: the final representation should enable us to formalize a selection of these techniques.
- The selected representation should enable us, besides the formalization of domain knowledge, to explicitate the difference between the domain knowledge and the knowledge resulting from a delta analysis. Since it is not likely that any existing knowledge representation technique makes it possible to notate differences in analysis results at the level of a domain analysis, we will have to define this "reuse relation" on the analysis level ourselves. This implies the construction of an "inheritance mechanism" at the level of analysis.
- Where the above criteria handle the expressive power of the knowledge representation to be selected, the third aspect will focus on the complexity of the algorithms needed to construct the active link based on this knowledge. Predicate logic, for example, demands less complex algorithms than frame-based techniques, which are in their turn more expressive.

To realize the active link between domain models and frameworks, the selected knowledge representation will thus enable us to explicitate domain knowledge, deltas and the relation between them, in a way that they will become suitable as a knowledge base for expert system technology.

2.2 A Logic Representation for Frameworks

When a suitable knowledge representation for domain knowledge and deltas is available, a logic representation for specific framework knowledge (e.g. design, class collaboration and reuse knowledge) needs to be selected.

Until now, this kind of framework knowledge was (insufficiently) supported by patterns and cookbooks ([8], [7], [12]), the main shortcoming being the informality of these techniques. Nevertheless, we will start from the existing co-evolution of design and implementation (see section 1.1). In this context we aim to extend the existing structural relations of the entities of a framework and its instances with the structures resulting from the analysis phase, with explicit references to the corresponding domain knowledge and deltas. For the moment, we consider these references as representations of the "why-relation": decisions made at the framework level have the analysis abstractions as their explicit reason. The explicitation of this relation will be a basis for the reasoning algorithms that will realize the active link between domain model and framework implementation.

When this active link is constructed, a co-evolution of the analysis phase and the existing co-evolution of design and implementation can be realized, implying that these three phases of the development cycle will evolve together whenever changes occur to one or more of them.

2.3 An Expert System for Domain and Framework Knowledge

Once the knowledge representations for domain knowledge and deltas, and for framework knowledge are selected, we plan to develop an expert system that uses these two kinds of knowledge as a knowledge base.

This expert system should support the following main activities:

- First of all, performing delta analysis based on
  - reusable domain knowledge that holds for the entire domain
  - earlier performed delta analyses
  - the logic representation of a framework and its reuse scenarios

  A suitable candidate system is a rule-based expert system that keeps the three points above variable while the following knowledge is contained in its rules:

  - general knowledge on framework reuse
  - general knowledge on delta analyses and the corresponding knowledge distillation
  - general knowledge on how the structure and reuse modalities constrain delta analyses

  A similar expert system will allow the analyst to perform a qualitative delta analysis that is driven both by the domain knowledge and by the structure and possibilities of the general framework. The selected expert system algorithmic will probably be a combination of forward- and backward chaining algorithms.

- The second main activity handles the planning of reuse scenarios based on the delta analysis. Recent research at PROG ([6]) increases our expectations significantly. Currently, we are able to assist the reuse of a framework to the level of adding classes and methods.

- Finally, the expert system technology should be iterative: instantiating the framework does not only imply the planning of a reuse scenario but also an interactive (while the framework developer is planning) update of the logic knowledge of this extended framework. This demands a third component that is able to (help to) extract and/or distill the structural knowledge from the extended framework to ensure that this knowledge is up-to-date to support the first main activity in the future. Machine learning techniques will possibly assist this activity.

3 Current State

At this early stage the main concern is the identification of a suitable knowledge representation for domain models and the results of the corresponding delta analyses.

A first step was to identify domain and delta analysis and their results, and to demarcate definitions for these concepts. It is our experience that most vertical market companies follow their own conventions when performing domain and delta analyses. Unfortunately, all these conventions seem to differ intensively, which adds another difficulty to the identification of a knowledge representation and which has slowed down the foreseen progress. However, we plan to construct a definition that is specific enough to be transformed into an existing knowledge representation, and in the same time general enough to cover the conventions of most companies that perform domain and delta analysis.

Once definitions of domain models, delta analyses and their components are decided, we plan to construct a logic program for domain models. This program - a kind of database - will mainly consist of facts that describe the roles of the different elements of a domain model and the relations between them. Based on the logic facts mentioned above, different views of domain models can be constructed. For example, one view might display all elements or a specific group of elements of the domain model, while another view presents all domain elements an element is related to and how these need to be adapted when changes on the first element occur.

At the moment, we are also investigating how related research can contribute to this first phase of our research. A previous attempt to explicitate domain knowledge with the purpose of facilitating framework development resulted in the concept of knowledge graphs [1], which showed to reduce the necessary amount of iterations to develop a robust and stable framework significantly. The main difficulty here turned out to be the need for specific inheritance semantics for certain knowledge domains, since the classical inheritance mechanism of the object-oriented paradigm is not always suited to model generalization and specialization relations in these knowledge domains. To solve this and related difficulties, the authors [1] propose to provide each software artifact with its own "intelligence" by means of a set of rules, ensuring the artifacts to actively keep the entire piece of software correct and up-to-date. Since this is exactly one of the main goals of the co-evolution research ([5], [11], [4], [6], [10]) mentioned above, a combination of co-evolution and knowledge graphs [1] may produce fruitful results for the formalization of domain models in the context of co-evolution of analysis, design and implementation.

4 Further Work

On a short-term basis, we plan to a finalize a suitable definition of domain models, delta analyses and their components. Next, these concepts will be coded into logic facts enabling us to develop different views on domain models. The long-term goal consists of the selection of a logic representation of framework - an extension of the existing co-evolution research (see section 1.1) - and the development of an expert system as described in section 2.3.

5 Conclusion

With the purpose of making the current framework development and reuse methodologies more structured and to avoid the problems caused by the implicit nature of the relation between domain analysis and framework implementation, we propose to construct a bidirectional, concrete, active link between domain models and frameworks. Our approach will apply existing A.I. techniques to realize a coupling between the (domain) analysis level and the framework implementation level. Starting from the existing co-evolution research of design and implementation, we aim to realize a co-evolution of analysis, design and implementation, implying all three phases evolve together whenever changes occur to one or more of them.

References

[1] Mehmet Aksit, Bedir Terkinerdogan, Francesco Marcelloni, Lodewijk Bergmans, *Deriving object-oriented frameworks from domain knowledge.*

[2] Wim Codenie, Koen De Hondt, Patrick Steyaert, Arlette Vercammen, *From custom applications to domain-specific frameworks*, Communications of the ACM, 1997.

[3] Krzysztof Czarnecki, Ulrich W. Eisenecker, *Generative programming*, Addison-Wesley, 2000.

[4] Kris De Volder, *Type-oriented logic meta-programming*, PhD thesis, Programming Technology Laboratory, Free University Brussels, 1998.

[5] Theo D'Hondt, Kris De Volder, Kim Mens, Roel Wuyts, *Co-evolution of object-oriented software design and implementation*, TACT Symposium Proceedings, Kluwer Academic Publishers, 2000.

[6] Sofie Goderis, *A reflective forward-chained inference engine to reason about object-oriented systems*, Bachelors thesis, Programming Technology Laboratory, Free University Brussels, 2000.

[7] Ralph E. Johnson, *Components, frameworks, patterns.*

[8] Ralph E. Johnson, *Documenting frameworks using patterns*, Oopsla'92, 1992.

[9] George F. Luger, William A. Stubblefield, *Artificial intelligence, structures and strategies for complex problem solving*, Addison-Wesley, 1998.

[10] Kim Mens, *Automating architectural conformance checking by means of logic meta programming*, PhD thesis, Programming Technology Laboratory, Free University Brussels, 2000.

[11] Roel Wuyts, *A logic meta-programming approach to support the co-evolution of object-oriented design and implementation*, PhD thesis, Programming Technology Laboratory, Free University Brussels, 2000.

[12] Inc. ParcPlace Systems, *VisualWorks cookbook*, 1994.