TABLE OF CONTENTS

TABLE OF CONTENTS 1

1 INTRODUCTION 3

1.1 Motivation 3
1.2 Current research in software architecture evolution 4
1.3 Reuse Contracts as a framework for evolution 4
1.4 Approach and contributions 5
1.5 Organisation of the dissertation 6

2 REUSE CONTRACTS 7

2.1 Conceptual level 7
2.1.1 Example 8
2.1.2 Inconsistencies between parallel evolutions 10
2.1.3 Applicability conflicts 10
2.1.4 Evolution conflicts 11
2.2 Formal representation 13
2.2.1 Foundations 13
2.2.2 A formalisation of reuse contracts 15
2.3 Implementation 19
2.3.1 Extension mechanisms 20
2.4 Summary 21

3 SOFTWARE ARCHITECTURES 23

3.1 Software Architectures 23
3.1.1 Motivation for study and analysis of software architectures 24
3.1.2 Architectural descriptions 25
3.1.3 Architectural styles 26
3.2 Architecture Description Languages 27
3.2.1 Domains of concern of ADLs 28
3.3 Evolution of software architectures 28
3.3.1 Execution-time evolution 29
3.3.2 Specification-time evolution 31
3.4 This dissertation 32
3.5 Summary 33

4 ADELA: A SIMPLE LANGUAGE TO DESCRIBE SOFTWARE ARCHITECTURES 34

4.1 Some characteristics 34
4.2 Architectural conceptual framework 34
4.2.1 Example: SOUL's architectural description 36
4.3 Syntax 38
4.3.1 Naming and scoping 39
4.3.2 Example: SOUL's architectural description 39

4.4 Semantics 41
4.4.1 Informal semantics of the increments *42*
4.4.2 Desired semantics *42*
4.4.3 Example: SOUL's architectural description *44*
4.4.4 Definition of Adela in term of reuse contracts *45*
4.5 Implementation 50
4.5.1 Semantics predicates *50*
4.5.2 Domain specific information module *52*
4.6 Summary 53

5 EVOLUTION OF SOFTWARE ARCHITECTURES 55

5.1 Evolution of architectural descriptions 55
5.1.1 Conflicts in evolution of architectures *57*
5.2 Applicability Conflicts 59
5.3 Evolution Conflicts 65
5.4 Implementation 70
5.4.1 Applicability conflicts *70*
5.4.2 Evolution conflicts *70*
5.5 Summary 71

6 EXTENSIONS TO THE LANGUAGE ADELA 72

6.1 Different kinds of extensions 72
6.1.1 Syntactical extensions *73*
6.1.2 Defining and detecting more conflicts *76*
6.1.3 Enriching the architectural model *77*
6.2 Reuse of architectural descriptions 81
6.2.1 Modelling Reuse *81*
6.2.2 Reuse versus evolution conflicts *82*
6.2.3 New language features *82*
6.2.4 Styles *83*
6.3 Summary 83

7 CONCLUSIONS AND FUTURE WORK 84

7.1 Summary 84
7.2 Future work 85
7.2.1 Reuse Contracts *85*
7.2.2 Definition of new evolution conflicts *86*
7.2.3 Adela as a change manager *86*
7.2.4 Runtime evolution *86*
7.2.5 Reuse of architectural descriptions *86*
7.2.6 Adela integrated in a traceability environment *87*
7.3 Main Contributions 87

8 REFERENCES 88

1 INTRODUCTION

1.1 Motivation

As software grows in complexity and size, software engineers recognise the need for more abstract descriptions that help in understanding the software, building and maintaining it. Software architectures are considered to be the natural evolution of design abstractions [Garlan&Shaw96].

There is no consensus about a definition of a software architecture. One of the first definitions is the one of Perry and Wolf, which states that architecture is concerned with 'the selection of architectural element, their interactions and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design' [Perry&Wolf92]. Nowadays, however, the most accepted and referenced definition is the one given by Garlan and Shaw, who define it as '(the level of design that) goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include the organisation of a system as composition of components, global control structures; protocols for communication, synchronisation, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives' [Garlan&Shaw96]. The architecture of a system can be described by specifying components, connectors and architectural configurations, which are connected graphs of components and connectors that describe the structure [Medvidovic&Taylor97].

Evolution is a very important issue in software architectures [Perry&Wolf92]. Most of the time evolution is done in an unsupported, unstructured and unanticipated way, resulting in applications with badly structured architectures. As systems evolve in this way they become more and more resistant to new changes. This effect is known as architectural drift. However, this problem can be reduced by providing good documentation on the design of the architecture, and tools that provide support for evolution of architectural descriptions.

Architectures as reusable assets play an important role in software product lines. A product line is a set of products that together address a certain market segment and which are built from a common set of software assets [Svalhnberg&Bosch99]. The main objective in these initiatives is reuse, and ideally the architecture itself forms the basis for reuse. The same architecture is reused and adapted if necessary to the needs of different customers. As Svalhnberg and Bosch [Svalhnberg&Bosch99] emphasise, the evolution of architectures in this context presents particular problems. 'Software in product-line architectures, once developed, is subject to considerable evolution, since a constant flow of new requirements is present. These requirements originate from the feedback received from customers concerning existing products that are on the market and from new products that are to be integrated into the product-line. Since so many new requirements that need to be handled out, potentially, are conflicting, the typical way of dealing with this is to create *two independent evolution cycles*,

i.e. for each product, incorporating new requirements that are product-specific, and for the product-line architecture as a whole, incorporating new requirements that affect all or most of the products in the product-line.' The presence of these two evolution cycles generates conflicts, because inconsistencies occur when changes are merged into one architecture.

The problems generated by the existence of independent evolutions is not limited to product-line based development environments, but also appears in every collaborative environment, in which many persons can apply different modifications to the same architecture. There is an important need to support evolution in a consistent way. The aim of this thesis is to detect the conflicts that appear because of inconsistencies in the evolution of architectures, focusing on the problem of unexpected interactions in independent modifications.

1.2 Current research in software architecture evolution

There are two kinds of evolution of software architectures addressed in literature: specification-time evolution and runtime evolution. Most of the research work is focused on runtime evolution. The problem in that area is to ensure that the system will go on running appropriately during the course of an architectural change. This concern imposes certain restrictions on the kinds of evolutions, which are not applicable for specification-time evolutions. For example in some of the approaches for runtime evolution it is not possible to deal with unexpected evolutions, all the possible modifications have to be foreseen during the design of the architecture of the system. Moreover, complex formalisms are needed in order to be able to ensure that the system will not crash unexpectedly when the architecture evolves.

Our concern, more linked with reuse, is specification-time evolution. There are some architecture description languages that allow incrementally defining architectures out of other architectural descriptions. But evolution support for architectures is limited and often relies on the specific programming language that will be used to implement the system [Medvidovic&Taylor97].

As far as we know there has been done no research on the specific problem of interaction of independent evolutions. Characterisations of which kinds of changes can appear in product line environments exist, e.g. [Svalhnberg&Bosch99], but little work is focused on simultaneous evolution of architectures and the detection and definition of the conflicts that appear in that context.

1.3 Reuse Contracts as a framework for evolution

The ultimate goal of this thesis is to provide support for the evolution of software architectures, detecting possible conflicts that can be caused by independent but simultaneous evolutions. This is a very broad problem, and solving it is a very ambitious task, but we will use a well-established technique, i.e. reuse contracts, as a basis to tackle it.

Reuse contracts [Lucas97][Codenie&al97][Mens99] are a technique that aims at providing support for software evolution and reuse by explicitly documenting the way a modifier incrementally evolves a software asset. The reused (or modified) software asset makes the relationships inside the asset in which the reuser can rely explicit. In this way, the reuse contracts approach succeeds in detecting possible problems that occur when the evolutions proposed by two independent modifiers of a software asset are merged.

We claim that reuse contracts provide a good framework to document and reason about the evolution of software architectures, and detect conflicts that can appear as a result of the interaction of independent evolutions of a given architecture.

In the context of this statement, we will reformulate the goals of the thesis as:

- Application and customisation of the reuse contracts formalism to the specific area of software architectures. Verification of whether reuse contracts provide a good framework to detect and reason about evolution conflicts at that level.
- A critical analysis of shortcomings of the reuse contracts model when applied to software architectures. This work will serve as a partial validation of the formalism for reuse contracts developed by Tom Mens in his PhD dissertation [Mens99], and of the application of reuse contracts in other domains (i.e. software architectures) than those for which they were originally intended.

1.4 Approach and contributions

In order to accomplish the goals mentioned above, the following tasks had to be carried out:

1) Definition of a minimal architecture description language called Adela that allows us to express architectures in an incremental way. This language is based on operations that describe modifications on architectural descriptions, such as adding an architectural element or deleting it. There is a language construct that allows expressing parallel evolutions. These operations are mapped onto basic operations in the reuse contracts formalism.
2) Definition of conflicts:
   a) Definition of well-formedness constraints on architectures and preconditions for the architectural operations that allow to determine if the application will raise a conflict. These preconditions are the basis for the definition of conflicts generated by simultaneous evolutions.
   b) Definition of conflicts that appear as an unexpected interaction between two operations, when they are merged in the same architectural description.

   Given an expression of Adela, reuse contracts provide an engine to detect the occurrence of conflicts.
3) Analysis and definition of possible extensions to the original language. These extensions were inspired by the features supported by other architectural description languages found in literature, thus providing evidence to prove that the approach is general enough. Analysis of new operations needed to support these features and more specific conflicts that could be detected based on this more precise information.
4) Implementation of a prototype textual tool to detect conflicts. The input is an expression in the proposed architecture description language Adela that expresses the evolution of an architecture, and the output is the architectural description that was described by the expression together with the conflicts that occurred during the evolution. The conflict detection engine is based on the PROLOG implementation of the reuse contracts formalism developed by Tom Mens. Implementation of some of the refinements proposed, and analysis of the implementation of the others.
5) Experimentation and validation on a case study.

The main contributions of this work are a validation of the reuse contract approach as a general foundation to handle software evolution, and the definition of a framework in which it is possible to reason about independent evolutions of architectures. In particular, we define a set of conflicts that appear as a consequence of independent but simultaneous evolutions of architectural descriptions.

The result is a basic language that supports the evolution of software architectures and detects conflicts. The language can be extended with new operations and new conflicts.

The use of reuse contracts and more specifically of Tom Mens' formalism [Mens99] and implementation provided us with a well-established framework to reason about the interaction among different independent evolutions, in which it was easy to define and detect some conflicts. But because of some constraints in the extension capabilities of this formalism, we found that the implementation of certain operations and conflicts involved changing the formalism itself.

One important drawback we encountered is that at the moment the experiments for this dissertation were being done the implementation for reuse contracts could check two modifications against each another, but previous modifications could neither be remembered nor re-checked against new ones. This fact imposed severe restrictions with respect to managing reuse. A possible solution for this problem is proposed in Tom Mens' dissertation [Mens99] and it is analysed in Chapter 6.

Given that the architectural description yields a high level view of a system, providing support for evolution in this early step in the life-cycle is very significant, since the changes at this level could have a major impact on many other assets. Having support in the detection of these conflicts can save time and can help in working with a consistent representation of the architecture.

The conflict detection engine developed in this dissertation could be integrated to an intelligent editor tool of software architectures, which might warn the users about the conflict and could even propose alternative solutions.

1.5 Organisation of the dissertation

Chapter 2 introduces *reuse contracts* as a technique to support software evolution. The formalism for reuse contracts together with some aspects of the implementation of a framework for reuse contracts in PROLOG are presented in the same chapter. Chapter 3 is devoted to the main concepts of the broad area of *software architectures*, and focuses on the more specific area of evolution of architectures. Chapter 4 presents the *architecture description language Adela*, which was designed to handle architectural evolution. The architecture of the SOUL system is introduced in Chapter 4 as well. SOUL is used as a case study and example in Chapters 4 and 5. Chapter 5 discusses some of the different *conflicts* that appear as a result of unexpected interaction of simultaneous evolutions, conflicts that can be detected based on evolutions expressed in the language Adela and using reuse contracts as a conflict-detection engine. The language constructs provided by Adela are very elementary. In order to provide evidence that the approach is powerful enough in chapter 6 we explore several possible *extensions* to Adela, that make it more compact or more suited for the description of certain kinds of architectures. Also in Chapter 6 the topic of *reuse* is tackled, and a solution for the documentation of reuse is presented. Chapter 7 summarises, presents *conclusions* and discusses different directions of future work.

2 REUSE CONTRACTS

'Software evolution is widely recognised as one of the most important problems in software engineering. Despite the significant amount of work that has been done, there are still fundamental problems to be solved. This is partly due to the inherent difficulties in software evolution, but also due to the lack of basic principles for evolving software systematically' [IWPSE98]. Reuse contracts provides a general framework to facilitate software reuse and evolution of any software asset.

In this chapter we present reuse contracts, as it will be the technique we will use throughout this dissertation to handle the evolution of software architectures. We organised this chapter in three parts. In the first one, the concept of reuse contract and the main goals and ideas behind them are explained, the second part is devoted to the formalism developed by Tom Mens [Mens99], and the third part presents the PROLOG implementation of the formalism.

2.1 Conceptual level

The reuse contracts (RC) approach aims at providing support for reuse and evolution of software artefacts. The main contributions of this approach are [Mens99]:

- Reuse contracts can be used as structured documentation of reusable artefacts, and assists a software engineering in adapting (or reusing) the software artefacts.
- Reuse contracts encourage disciplined reuse and evolution without being too coercive.
- They assist in assessing how much work is necessary to update previously built applications that rely on the changed asset.

The approach is based on the explicit documentation of (otherwise) hidden assumptions of software artefacts and the modifications made to them by means of *contracts*. The essential idea is that reuse and evolution of software artefacts is based on an explicit *contract* between the *provider* of an artefact and a *modifier* (also called a *reuser* or an *evolver*) that incrementally modifies this artefact. In this contract, both the provider and the reuser have *contractual obligations*. The primary obligation of the provider is to document on which of its properties and assumptions reusers can rely. This is specified in a so-called *provider clause*, which contains all information that is important for modifiers, i.e. interface descriptions that partially document the internal structure of software artefacts [Lucas97]. On the other hand, the modifier specifies in the *reuser clause* how the artefact is actually being reused, or how the component evolves [Mens99].

To be able to express the specific way in which an artefact is reused, different kinds of evolutions need to be identified. *Contract types* encode the different ways an artefact can evolve. They are the only means of adapting the interface descriptions. In the reuse contracts model a basic set of contract types is provided. The most relevant are *extension*, to add new

elements to interface, *refinement*, to add extra dependencies to an interface, and their inverses, *cancellation* and *coarsening*. These types impose obligations, permissions and prohibitions on the reuser. Contract types are fundamental to disciplined reuse, as they form the basis for detecting conflicts when provided artefacts are modified [Mens99].

The software artefacts can vary depending on the domain in which reuse contracts are used. An artefact may be a single class [Steyaert&al96], a set of collaborating classes [Lucas97], a UML collaboration diagram [Mens&al99a] or any other evolvable component, as software architectures in our case. Reuse contracts were originally defined in [Steyaert&al96], later refined by [Lucas97] and generalised and formalised by [Mens99].

An example of conflict detection of reuse contracts is presented in the next subsection, and the different kinds of inconsistencies the approach aims to detect are explored in the subsequent three subsections.

2.1.1 Example

We will illustrate the main ideas of reuse contracts with an example taken from [Codenie&al97]. In this example, the software artefacts are classes that contain methods. A dependency between two methods A and B represents that in the implementation of method A there is a call to method B.

Consider a class TapeLibrary that represents a tape library of some television station. This class includes a method addTape to add a single tape and a method addTapes to add a set of tapes, and maybe some other methods. In the implementation of addTapes, addTape is called with each of the tapes we want to add as argument. This information is documented in a provider clause, which is graphically illustrated in Figure 1. The relationships between methods is written between brackets. For example addTapes () [addTape] means that the method addTapes calls the method addTape.

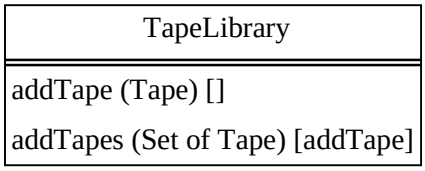

Figure 1. Tape Library Class

Now suppose that a new requirement appears so that the tape visioning department must be notified each time a tape is added to the library. This can be achieved by subclassing the TapeLibrary with the new class OOTVTapeLibrary, and modifying the method AddTape so that it calls a new method notifyVisioning.

This evolution consists of adding the method notifyVisioning (which corresponds to the contract type Extension) and adding an extra dependency from addTape to the notifyVisioning method, which corresponds to the contract type called *refinement*. This is illustrated in Figure 2.

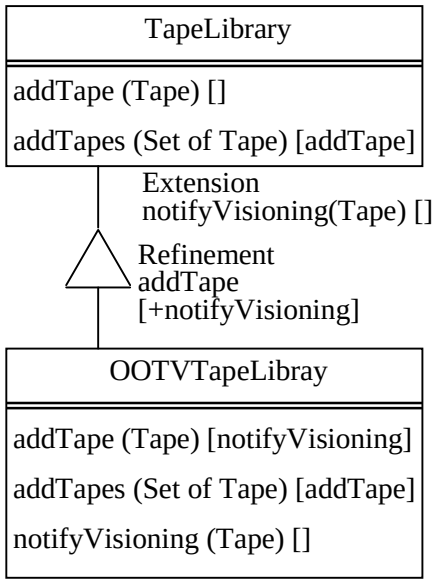

Figure 2. Evolution of Type Library Class (I)

Independent from this evolution, motivated, for example, by efficiency reasons, the implementation of the method addTapes is changed, so that it does not rely on addTape any longer. Instead, the adding of tapes is done explicitly by the addTapes method itself. Figure 3 depicts this evolution.

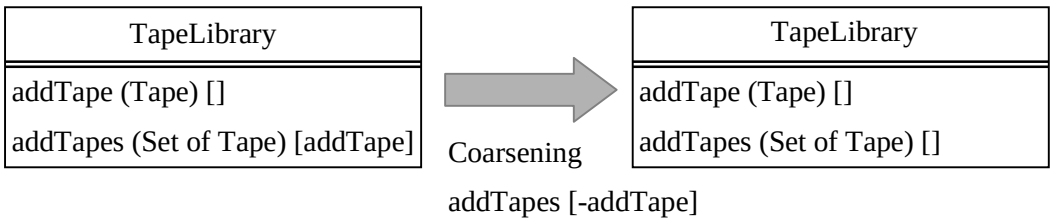

Figure 3. Evolution of Tape Library Class (II)

When these two evolutions are merged, we find that the implementation of addTapes does not call the notifyVisioning method through addTape as it was expected, so the tape-visioning department is not notified. The developers who made these evolutions should be informed about this situation so it can be resolved. In this case, this could either be done by making addTaped dependant on addTape again or by also making an explicit call to notifyVisioning in the changed addTapes method.

This is an example of one of the kinds of inconsistencies that reuse contracts aim to detect.

2.1.2 Inconsistencies between parallel evolutions

We will distinguish between two kinds of inconsistencies, *structural* or *syntactic* inconsistencies and *semantic* or *behavioral* inconsistencies.

Structural inconsistencies occur when two independent evolutions of the same artefact cannot be merged because a part of the artefact, which is required for one evolution, is modified by the other, and vice versa. So one of the modifications cannot longer be applied. The conflicts these inconsistencies generate are called *applicability conflicts*. An example of an applicability conflict is when one modifier removes a class and another one tries to add an operation to it.

But sometimes, even when two modifications of the same component can syntactically be merged, their combination can unnoticedly cause undesired interactions. These inconsistencies are called semantic and they are harder to detect. Since the modificatins can be legally merged, they are applied, but the result is not entirely what the evolvers expect. The conflicts generated by this kind of inconsistencies are called *evolution conflicts*. An example of an evolution conflict is the one given in the previous section.

A characteristic of evolution conflicts is that it is not possible to precisely distinguish between a real problem and an expected effect. In the example of the previous section, it may have been possible that the visioning center did not want to be notified when a set of films arrive (for example because in that case the notification should follow other procedures) and only wants to be notified in the case of a single film. Nevertheless it is valuable to warn about the occurrence of such situation. The developers can still decide what to do with the warning and whether there is a real conflict that needs to be fixed or not.

2.1.3 Applicability conflicts

Applicability conflicts are the ones that appear when two independent evolutions are such that the application of one of them inhibits the application of the second. For example, two users trying to add a class with the same name causes a *Duplicate Node* applicability conflict, because only one of the classes can be added. For their detection they rely on assumptions about the well-formedness of the system, such as that there can only be one class with a certain name.

Applicability conflicts can be detected by examining the contract types and their actual parameters. In the following table the different conflicts are displayed in relation with the operation that generate them[1]. The rows and columns represent operations (contract types), and in the intersection of a row and a column we can read the conflict that is generated when the two operations are applied to the same software artefact (graph) by different users.

[1] The operations in the table refer to nodes and edges because, as we will see in the next section, reuse contracts can be applied to any software artefact that can be mapped onto a graph.

Table 1. Table of conflicts

	Extension (v,u)	**Cancellation (v,u)**	**Refinement (e,v,w,t)**	**Refinement (e,u,v,t)**	**Coarsening (e,v,w,t)**	**Coarsening (e,u,v,t)**
Extension (v,w)	Duplicate Node					
Cancellation (v,w)		Double Cancellation	Undefined Source	Undefined Target		
Refinement (e,v,w,p)		Undefined Source	Duplicate Edge			
Refinement (e,u,v,p)		Undefined Target		Duplicate Edge		
Coarsening (e,v,w,p)					Double Coarsening	
Coarsening (e,u,v,p)						Double Coarsening

2.1.4 Evolution conflicts

We will present the approach to detect evolution conflicts adopted in Mens' dissertation [Mens99].

Evolution conflicts are detected in a way different from applicability conflicts. Each evolution conflict is related to a pattern that is likely to carry a problem. Given an evolution request, after applying it the graph is checked to see if any of the patterns that represent an evolution conflict is contained in it.

Evolution conflicts are different from domain to domain, in the sense that a scenario that may be a conflict in one domain is not necessarily a problem in a different one. Let's take as an example the occurrence of cycles. It is important to warn about the existence of cycles of method calls, because this may give rise to unexpected infinite recursion when invoking a method. But if the artefacts are software architectures a cycle of data flows is very natural and occurs quite often. In the later case it will probably be useless to warn about such cycles. Nevertheless we will enumerate the evolution conflicts defined by Mens, mostly in order to illustrate them [Mens99].

EC1: Reachability conflict

A reachability conflict occurs when two modifiers independently add two new relationships so that the element reached by one of them is the source of the other, as it is illustrated in Figure 4.

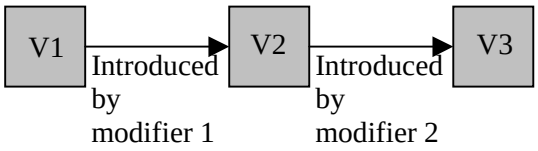

Figure 4. Reachability conflict

This situation may be a conflict in the case of classes and methods, because it may indicate that the method V1 is calling method V3 indirectly as an unexpected result of the interaction of that two evolutions, which may not have been intended by either of the modifiers.

EC2: Double Reachability Conflict

A double reachability conflict occurs when two modifiers independently add two parallel edges. This scenario, depicted in Figure 5, may be a problem because it may mean that both modifiers are adding the same interaction twice with a different name.

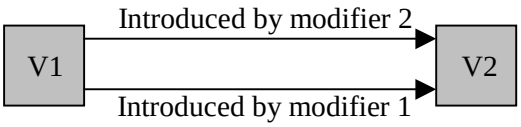

Figure 5. Double Reachability conflict

EC3/EC4: Double Source/Target Conflict

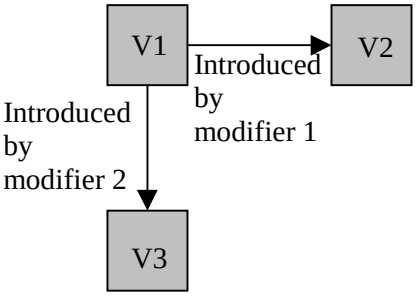

Figure 6. Double Source Conflict

Double Source conflicts occur when two modifiers independently add two relationships to the same source. This situation is illustrated in Figure 6. Double Target is similar but they both add two relationships with the same target.

EC5: Cycle Introduction Conflict

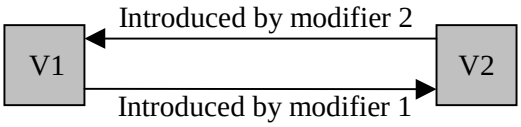

Figure 7. Cycle Introduction Conflict

A cycle introduction conflict is generated when two users independently add relationships that generate a cycle. As we said before, in the context of classes this will often be a problem because it would represent an infinite recursion because of two methods calling each other.

2.2 Formal representation

In his Ph.D. dissertation [Mens99], Tom Mens defined a formal foundation for reuse contracts. This foundation is domain-independent and scalable. Category theory is used as the underlying formalism. Software components are represented by labelled typed graphs and evolution is represented by conditional graph rewriting.

We will first present this formalism on top of which reuse contracts are modelled, and after that we will explain then the representation of reuse contracts themselves.

2.2.1 Foundations

In this section we will present the basic underlying ideas of the formal foundation, without going into technical details. We need to introduce some of these concepts because they establish the basis for the reuse contracts framework we will use in the rest of the dissertation. All the definitions are taken from Mens' dissertation [Mens99].

2.2.1.1 Labelled typed graphs

In the formalism proposed by Tom Mens, components are represented by *labelled typed graphs*, i.e. graphs that have a *label*, a *type* and a *set of constraints* attached to each node and edge. Labels allow distinguishing one node from another, or one edge from another. There cannot be two nodes with the same label. Types capture similarities among certain groups of nodes or edges. Types can be used to classify similar nodes and edges, by giving them the same type. Constraints allow attaching any kind of information to the nodes, which can be used later in order to detect conflicts. For example, the identification of the last modifier that made changes on the node (or edge) can be stored in the constraints.

If we take the example of classes as software artefacts, the different classes and methods of a system would be represented as nodes, and method invocations would be represented as edges. The labels would be the names of the classes or the names of the methods. *Class* and *operation* would be different types of nodes, while *methodInvocation* would be a type of edge. The constraints in a class node will contain information such as who was the last developer that modified the class.

On top of labelled typed graphs a *nesting mechanism* is built, that can be used as an encapsulation and layering mechanism. This is the mechanism that allows expressing that the methods belong to the class: we will say that they are nested inside it.

Even if in his dissertation Tom Mens nesting is defined as a relation on the nodes of the graph, at the time the experiments for this dissertation were being done nestings were expressed by means of a special kind of edge type called *nested*. So in the example of classes, if we want to express that the methods are nested inside the class that defines them, then these methods should be related with their class by means of a nesting edge.

2.2.1.2 Types

Types allow defining categories of nodes and edges. We can define constraints for the types, and in this case all the nodes of that have to satisfy constraints specified.

Node types and edge types are organised in a hierarchy. This hierarchy is formally expressed as a partial order over types, which we will call subtyping. The constraints we attach to a type will be inherited by all its subtypes. Some constraints can be expressed by means of a *type graph*, which shows restrictions on the types of the nodes that edges can connect.

In our example, the type graph would be one represented in Figure 8.

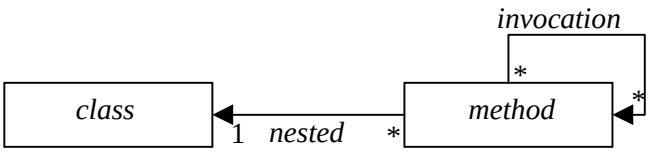

Figure 8. Type Graph

This type graph expresses that edges of type nested can only relate nodes of type *method* with nodes of type *class*, in a one-to-many relationship, that edges of type *invocation* can only relate methods with other methods in a many-to-many relationships.

There are some constraints on types cannot be expressed in a type graph. Suppose we wanted to differentiate between internal method calls, where a method calls another method in the same class, and external method calls, where a method in one class calls a method in a different class. There could be two subtypes of *methodInvocation*, *internalInvocation* and *externalInvocation*. A constraint on *internalInvocation* edges is that the class to which both the source and the target method of the edge belong should be the same. This constraint cannot be expressed by means of the type graph.

2.2.1.3 Graph rewriting systems

An evolvable software artefact is modelled as a *graph rewriting system*. A graph rewriting system has an *initial graph* and a *set of productions*. Starting from an initial graph the set of primitive productions (evolution operations) can be used to transform (evolve) the graph into a new version.

A conditional *production* P basically consists of a rule that states that a part of a graph can be replaced by another graph given certain conditions. Figure 9 depicts the graphical representation of a production. The one in the picture is a conditional production called

Refinement that adds an edge named e of type *t* between two nodes v and w provided that those nodes exist and there is no edge e between those nodes yet.

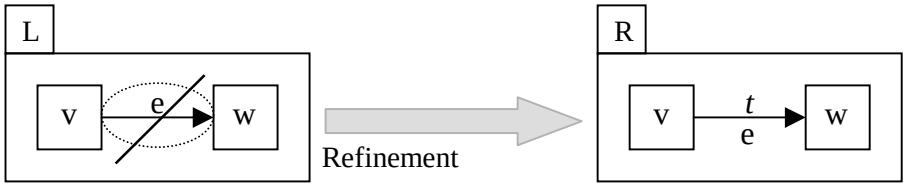

Figure 9. Refinement Production

As it is depicted in the picture, a production has a left-hand side L and a right-hand side R. Applying a production P to a given graph G is replacing an occurrence of its left-hand side L in G by the right-hand side R. The production also has an embedding transformation *embed* that specifies the details of how the right-hand side should be inserted in relation with the edges of G that arrive in or leave from nodes of L. If L and R are graphs, and P is a production, then we write P: L→R. A conditional production consists of a production and a set of preconditions and a set of postconditions that need to be satisfied before and after the application. In the example of Figure 9 there is a precondition that states that edge e does not exist before the application, and a postcondition that states that e exists after the application.

The subgraph L of G is called an *occurrence of L in G*, or a *match* for production P. The match is the part of graph G that will actually be replaced or modified.

The application of a production P: L→R to a given graph G is called a *direct derivation*, and represents one step of evolution.

One important notion is parallel independence. Two direct derivations (starting from the same initial graph G_0) are parallel independent if they can be applied in any order with the same result. This implies that none of the derivations delete or introduce nodes or edges needed by the other, meaning that they do not interact and that they do not produce conflicts. An example, given a graph G that has nodes m and n of type *w*, the two derivations corresponding to the productions refinement(e,n,m,*t*) and cancellation(n,*w*) are not parallel independent, since the first one adds an edge to the node the second one deletes.

2.2.2 A formalisation of reuse contracts

On top of labelled graphs and graph rewriting the reuse contracts are modelled, together with the conflicts that appear as unexpected interactions in independent evolutions.

2.2.2.1 Modelling reuse contracts

The concepts we presented in subsection 2.2.1 will be used in this section to model reuse contracts.

- The *provider clause* in a reuse contract will be represented by a labelled typed graph G.
- The kind of modification that occurs, namely the *contract type*, corresponds to a graph production. In the previous subsection we saw the graphical definition of the production corresponding to the contract type Refinement.

- The *modifier clause*, which expresses the actual details of the modification, is defined by a match that identifies how the contract type is applied to the provider clause. Remember that a match is the part of the graph under evolution (the provider clause) that 'matches' the left side of the production. The modifier clause can be thought of as containing the actual parameters of the contract type for that specific modification.
- A *reuse contract* itself is represented by a graph derivation that involves the production and the match.

2.2.2.2 Contract Types

In the formal model two contract types were added to the four basic ones (Extension, Refinement, Cancellation, Coarsening) in order to deal with types: NodeRetyping and EdgeRetyping. They allow changing the types of nodes and edges.

All the contract types are defined in terms of conditional productions. They all have application preconditions and postconditions attached. Figure 10 and Figure 11 show the conditional productions for each of the contract types in graphical notation.

The graphical notation may need some explanation. The squares represent nodes and the arrows represent edges. The presence of an edge (or a node) in the graphic means that the edge (or node) is required in the graph. A crossed out edge (node) means that the presence of that edge or node is forbidden in the graph. Labels are always indicated. Types are omitted when they are not needed, and they are in italic font.

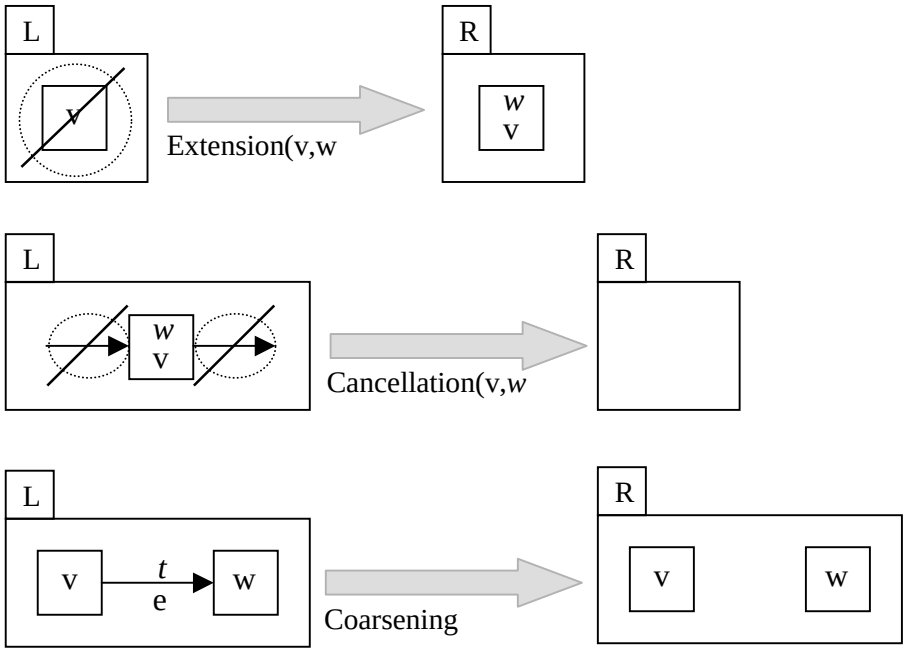

Figure 10. Productions associated to the contract types (I)

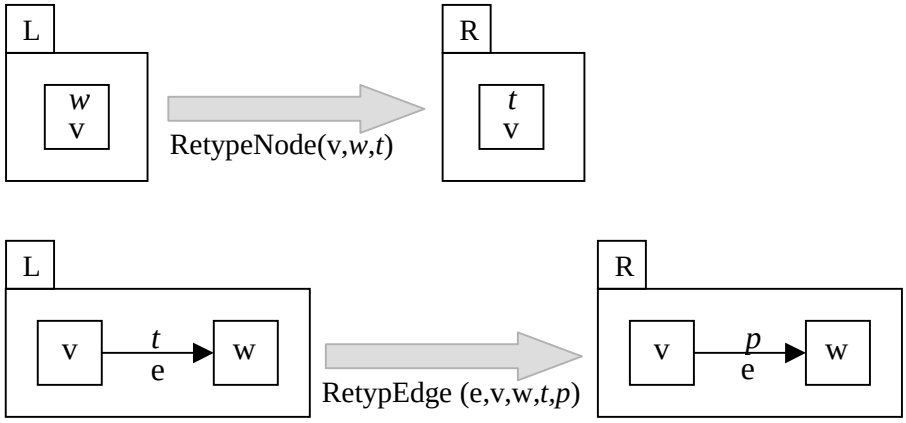

Figure 11. Productions associated to the contract types (II)

2.2.2.3 Detecting inconsistencies and conflicts

Applicability conflicts are defined formally in terms of properties of the productions (contract types) involved.

Two reuse contracts with productions (contract types) P1 and P2 lead to an applicability conflict if P2 is not applicable after P1, or if P1 is not applicable after P2 in the graph. The following property allows checking for conflicts:

> Two primitive reuse contracts $G \Rightarrow_{P1} G_1$ and $G \Rightarrow_{P2} G_2$ lead to an applicability conflict if and only if $\exists\, C_2 \in \text{PreCond}(P_2)$ such that C_2 is not satisfied in G_1, or $\exists\, C_1 \in \text{PreCond}(P_1)$ such that C_1 is not satisfied in G_2.

Evolution conflicts are detected in a different way. Recall that they are not detected looking at the contract types, as is the case for applicability conflicts, but by looking at the graph itself. A set of graph patterns can be defined, where each pattern corresponds to an evolution conflict, i.e. a situation that is likely to be a problem. Conflict detection then comes down to searching for these patterns in the graph resulting from application of the evolution. If the pattern is present in the graph (up to a match) then an evolution conflict exists.

> Let EvolConf be the set of all possible evolution conflicts patterns. Let $G \Rightarrow_{P1} G_1$ and $G \Rightarrow_{P2} G_2$ be two contract types that do not have applicability conflicts. $G \Rightarrow_{P1} G_1 \Rightarrow_{P2} H$ leads to an evolution conflicts if $\exists\, C \in$ EvolConf: $\exists$ injective match $m: C \rightarrow H$.

The evolution conflict patterns are the ones we presented graphically in section 2.1.4. Each node and edge saves in the constraints the information about the last modifier. Each of these situations is considered a conflict only if the modifiers stored in the constraints of the evolved nodes or edges are different.

A strong limitation of this approach is that it is possible to detect conflicts based only upon the nodes and edges that are present in the graph. It is not possible to detect conflicts based on the entities that were deleted. To illustrate this problem, consider the example presented in section 2.1.1. The problem was generated because of the interaction of a Refinement with a Coarsening. If we try to detect this conflict looking only at the resulting graph, we will not be able to do it, because in the graph we do not have the information about the Coarsening that took place! The coarsening just removed some dependencies from the graph, but it is

impossible to know which ones afterwards. The solution is to document the coarsenings and cancellations in the graph by means of a new type *removed*, that is attached to removed nodes and edges. So instead of actually deleting an entity, we mark it deleted. Now it is possible to detect the evolution conflict in the example looking for the following pattern depicted in Figure 12 in the resulting graph.

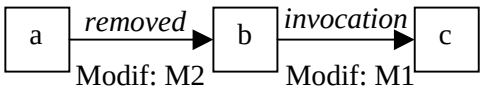

Figure 12. Conflict pattern

2.2.2.4 Composite contract types

Since the primitive contract types are very simple, it is necessary to have a mechanism to define more complex contract types out of other contract types. A *composite contract type* is a sequence of contract types applied one after the other.

The set of applicability conflicts a composite contract type can generate is a subset of the union of the sets of conflicts that each of the contract types that take part in it can generate. The reason is that some intermediate conflicts generated by contract types at the beginning of the sequence can be solved by the application of the contract types that appear later in the sequence.

As opposite to the applicability conflicts, a composite contract type can give rise to extra evolution conflicts, since it can capture more precisely the intention of the evolvers. We will see some examples of this in Chapter 5.

Some composite contract types are predefined. We will mention some of them and give a brief description:

- RedirectTarget(e,u,v,t,w) changes the target of an edge e. u is the old target and w the new one.
- RedirectSource(e,u,v,t,w) is similar to RedirectTarget.
- ConnectedExtension(u,w,t,[(e_1, $_{v1}$),(e_2, v_2),….,(e_n, v_n)]) adds a new node u of type w to the graph, and a set of edges, all of them of type t. e_i, $n \geq i \geq 1$ are the names of the edges, while v_i are the target nodes.
- MoveNode(d,w,a.b,a.c) moves the node labelled d from the node a.c to the node a.b.
- RelabelNode(u,w,v) changes the label of node u of type w to the new label v.

New conflicts appear as a consequence of the interaction of some of these new contract types with the old ones. We will see examples of conflicts introduced by MoveNode in chapter 5.

2.2.2.5 Documenting reuse and evolution

A problem that is still not solved with this approach is how to solve the conflicts that arise due to the interaction of reuse and evolution. Suppose we have a software artefact that is reused several times in the software system, and all these reuses are documented by means of reuse contracts. Now suppose that the original software artefact is replaced by an evolved version. We want to be able to detect the conflicts that appear with respect to the different reuses.

This is the scenario that appears when a framework is customised for different customers and at the same time it evolves to a better version.

Note that currently it is not possible to solve this situation because the different adaptations are forgotten once they are performed, only the resulting graph remains. Some extensions to the formalism need to be done, so that the reuses are explicitly documented in the graph. Mens proposes to use edges to model evolution information, placing the reuse contract information in the constraints of the edge. Reuse edges and evolution edges could be differentiated by using different types [Mens99]. In order to detect conflicts, each time a modification is done in the graph it is documented by a new edge of the corresponding type and conflicts are checked against all the other edges that express reuse or evolution of that graph.

2.3 Implementation

Tom Mens developed a prototype framework for reuse contracts in PROLOG. In this prototype the graph is described by a set of facts that represent the nodes and edges. The structure of the facts is as follows:

```
node(GraphName, NodeName, Type, Constraints)
edge (GraphName, EdgeName, SourceNodeName, TargetNodeName, Type, Constraints)
```

The name of the node uniquely identifies it in the graph, and similarly for edges.

The graph can be modified by means of contract types. The contract types extension, cancellation, nestedExtension, nestedCancellation, refinement, coarsening, retypeEdge and retypeNode and have the following syntax:

```
extension(NodeName,Type),
nestedExtension(NewNodeName, NewType, OldNodeName, OldType),
cancellation(NodeName,Type),
nestedCancellation(NewNodeName, NewType, OldNodeName, OldType),
refinement(EdgeName, SourceNodeName, TargetNodeName, EdgeType),
coarsening(EdgeName, SourceNodeName, TargetNodeName, EdgeType),
retypeEdge(EdgeName, SourceNodeName, TargetNodeName, OldType, NewType),
retypeNode(NodeName, OldType, NewType).
```

The set of preconditions for each of these modifications is defined by means of the predicate preconditions. As an example, the preconditions for the contract type coarsening are:

```
preconditions(production(coarsening,[E,V,W,T]),Graph) :-
  presentNode(Graph,V),
  presentNode(Graph,W),
  presentEdge(Graph,E,V,W,T).
```

They express that the target and source nodes and the edge to delete have to exist in the graph.

Well-formedness constraints for edge types are expressed by the predicate edgeTypeConstraints.

Two main predicates are provided that enable to express evolution on a graph: evolve and merge. Predicate evolve applies a modification to a graph, while merge merges two modifications and applies them together to the graph.

The evolve predicate receives the name of the graph to be modified, a modification and the name of the user that proposed it. It returns a (possibly empty) set of conflicts and the updated graph. For example,

evolve(G, extension(N,*node*), Evolver, Conflicts)

adds a new node called N to graph G and remembers that evolver is the person that fired the modification, if the preconditions for the application of extension(N,*node*) are satisfied in N. If they are not satisfied, it returns a conflict.

First evolve checks if the preconditions are satisfied, if they are not satisfied it will return an applicability conflict. Then it performs the modification and finally it checks if the resulting graph satisfies the well-formedness constraints associated with the types, if they are not a conflict will be returned. It returns the resulting graph and the set of conflicts found.

The merge predicate receives two modifications, the names of the users that proposed them and a graph. It returns a set of conflicts and the updated graph. For example,

merge(G,extension(N,*node*),cancellation(M,*node*),Evolver1,Evolver2,Conflicts)

adds a new node N and deletes node M from the graph G, remembering the names of the persons that fired the modifications, if the preconditions of the operations are satisfied in G and if there is no unexpected interaction between the modifications. Otherwise it returns a conflict.

It applies the modifications one at a time, checking if the preconditions are satisfied before applying each of them, and checking if the well-formedness constraints are satisfied after each of them. If the preconditions or well-formedness constraints are not satisfied for any of the modifications it raises an applicability conflict, and checks the table of conflicts (an implementation of the table of conflicts Table 1) in order to be able to specify which specific conflict was generated. If no applicability conflict is generated, after both modifications have been applied, it checks if one of the evolution conflict patterns is present in the resulting graph.

This framework provides some extension mechanisms that allow customising it to a specific domain. In the following subsection we will explore some of them.

2.3.1 Extension mechanisms

New types, new operations and new conflicts can be added in order to adapt the framework to a specific domain. We will see examples of the practical use of most of these extension mechanisms in the Implementation sections of Chapter 4 and Chapter 5.

Although the trivial extensions are very easy to make (such as renaming a conflict, or adding a precondition for an operation), in order to make more complex extensions (such as defining new evolution conflicts) it is necessary to have a fairly good insight in the implementation of the whole framework.

2.3.1.1 New types

Types are organised in a hierarchy with multiple inheritance, by means of the predicate subtype. The types *node* and *edge* are the roots of the hierarchy. New types can be added in by extending the type hierarchy. In order to add a new type *type1* subtype of type *node*, for example, the fact

subtype(*node*, *type1*)

has to be added.

Constraints on edge types are defined by means of a predicate called edgeTypeConstraints. The constraints for new edge types can be expressed by simply adding a new rule for this predicate.

2.3.1.2 New preconditions

New preconditions can be added in order to restrict the existent or new operations by adding clauses to a predicate domainSpecificPreconditions. This predicate is called from the evolve and merge predicates, before the translation is performed.

2.3.1.3 New operations

New operations can easily be defined if they can be translated into one basic contract type by means of the predicate translate-contract-type. This predicate is automatically called by the predicates evolve and merge, so once the translation is defined there is no need of further customisation. The preconditions for the domain specific operations are specified by defining the predicate domainSpecificPreconditions.

In the current implementation there are no mechanisms to define compound operations, but the implementations of such mechanisms is not very difficult.

The implementation and integration with the rest of the framework of those operations that cannot be defined in terms of the provided contract types is more complex.

2.3.1.4 New conflicts

The applicability conflicts can be renamed into names that are more significant to a specific domain by means of the translate-conflict predicate. New applicability conflicts can be detected by extending the domainSpecificPreconditions predicate. When a new operation is defined, the interaction with all the existent ones has to be analysed in order to update the table of conflicts. Th table of conflicts can be extended in a modular way, by adding clauses to the table predicate.

New evolution conflicts can be added by defining new conflict patterns.

2.4 Summary

In this chapter we presented reuse contracts, a technique that aims at providing support for software evolution of object oriented software artefacts. We illustrated the kind of conflicts this techniques helps in detecting with an example.

We also introduced the formal foundation of reuse contracts developed by Tom Mens and its prototype implementation in PROLOG. The formalism is general, it does not address any specific kind of software artefact in particular, but anything that can be represented as a graph. In order to apply it to a specific domain, the software artefacts in that domain have to be expressed as graphs and their evolutions as contract types. The conflicts can be customised to those in the specific domain as well. The prototype implementation provides two predicates to deal with evolution: one of them that applies a modification to a graph and the

other one merges two different evolutions and applies them to the graph, detecting the conflicts that appear because of the interaction of the two proposed evolutions.

We consider that reuse contracts are an appropriate technique to be applied in the specific area of software architectures.

3 SOFTWARE ARCHITECTURES

Garlan and Shaw state that 'as the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation'. Structural issues that involve the whole system can and should be expressed in the software architecture level of design [Garlan&Shaw96].

In this chapter we will introduce the concept of software architecture, justify the importance of that view of the system and explore some of the research that has been done in that area. We will which essential features that need to be supported by a language to describe architectures. Different architecture description languages (ADL) are suited for different concerns. We will explore the different concerns addressed but ADLs in literature, focussing on the concern of evolution of software architectures. We will finish this chapter situating the dissertation in relation with the other works in the area.

3.1 Software Architectures

Software architectures focus on the overall system structure, which includes 'the organisation of a system as composition of components, global control structures; protocols for communication, synchronisation, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives'[Garlan&Shaw96].

Informal architectural descriptions have been used by practitioners for a long time, expressed as box-line drawings accompanied by terms such as 'client-server organization' or 'layered system'[Allen&Garlan97]. But it is mainly in the '90s that many researchers have focused on general support for the process of developing architectures [Perry&Wolf92].

The architecture of a system involves the description of the elements that constitute the system, the interactions among them (often called connectors), the patterns that guide their composition and constraints about the patterns [Garlan&Shaw96].

A feature that distinguishes of software architectures from other views of the system is the explicit modelling of connectors [Oreizy&al98]. While components are the computational entities, connectors mediate and govern interactions among components. In this way connectors separate computation from communication, minimise component interdependencies and facilitate system understanding, analysis and evolution [Oreizy&al98].

The role of architectures in the software development cycle is still under discussion. In 1992 Perry and Wolf characterised the different phases of a software product by the kind of entities, properties and relationships of those entities, that are important for that phase in the following way:

- *Requirements* are concerned with the determination of *information* and processing together with the characteristics of that information and processing needed by the user of the system.
- *Architecture* involves the selection of *architectural elements*, their *interaction* and *constraints* on those elements to provide a framework in which to satisfy the requirements and serve as a basis for the design.
- *Design* is about modularization and detailed interfaces of *design elements*, their *algorithms* and *procedures*, and the *data types* needed to support the architecture
- *Implementation* addresses the *representation of algorithms and data types* that satisfy the design, architecture and requirements.

In other words, they consider the architecture as a phase distinct from design that deals with different entities. Medvidovic and Rosemblum discuss the relationship between design and architecture. They mention that in current literature three positions are taken:

- Architecture and design are the same.
- Architecture is a level of abstraction above design, it is another step in software development.
- Architecture is something new and different from design.

According to them all three are partially correct. The explicit focus on connectors and components distinguish architectures from traditional design. However, as an architecture is refined the 'connectors lose prominence by becoming distributed across the lower level architecture's elements'[Medvidovic&Rosenblum97].

3.1.1 Motivation for study and analysis of software architectures

Perry and Wolf enumerate some of the benefits of the explicit specification of architectures [Perry&Wolf92]. An architectural specification should allow to:

- Prescribe the architectural constraints at the desired level, being able to express at the architectural level only what is needed at that level.
- Separate aesthetics from engineering.
- Express different aspects of architecture in an appropriate manner.
- Perform dependency and consistency analysis, between architecture, requirements and design, and between different parts of the architecture.

The study of software architectures and support for their description and analysis has severe consequences in the software development process. Perry and Wolf state that evolution and customisation of software architectures are two important factors in the high cost of software. [Perry&Wolf92]. As a system evolves, it becomes more resistant to change. This effect is caused by architectural drift due to violations of the original the architecture, mostly because 'the software engineer is ignorant of an imposed software architecture due to little or no documentation on the subject' [DeHondt98]. With good documentation on the intentions of the software architect in the design of the architecture, and good tools that provide support for change this would be a much less significant problem. The other factor is customisation. In an ideal situation, for the standard parts of a system the architect can select from a set of well known and understood elements and use them in ways appropriate to the desired architecture.

'The use of standard templates for architectural elements frees the architect to concentrate on those elements where customisation is crucial.' [Perry&Wolf92] The use of architectural styles, which will be defined later, is the proposed solution. ADLs should help in the definition and evolution of styles, as well as in the process of adapting the styles to a particular system.

In the next subsequent sections we will specify which are the essential features needed to model software architectures, and we will define the concept of architectural style.

3.1.2 Architectural descriptions

As we said before, components, connectors and architectural configurations are the building blocks of architectural descriptions. In this subsection we will describe what they represent in more detail.

- *Components*. The components are the units of computation and data store. A component can be as small as a single procedure or as large as an entire application. Each component may have its data and/or execution space or it may share them with other components, and may refer to one compilation or execution unit in the implementation or to many of them [Medvidovic&Taylor97]. Components can be clients, servers, filters, layers, databases [Garlan&Shaw96]. An entire architectural description may in turn be used as a composite component in a larger system [Garlan&Shaw96].
- *Connectors*. The connectors are 'building blocks used to model interactions among components and rules that govern those interactions'. A connector may or may not correspond to a compilation unit in the implementation. They may be 'separately compilable message routine devices or shared variables, table entries, buffers, instructions to a linker, dynamic data structures, procedure calls, initialising parameters, client-server protocols, pipes, etc' [Medvidovic&Taylor97].
- *Architectural configurations*, also called topologies, are connected graphs of components and connectors that describe architectural structure. They provide the information to determine whether the connections between components and connectors is consistent with the specifications of both parts, whether their interfaces match, and whether the combined semantics results in the expected behaviour. They are important in the analysis of concurrent and distributed aspects of an architecture, e.g. 'potential for deadlock or starvation, performance, reliability, security, security, etc'[Medvidovic&Taylor97].

An example of an architectural description is illustrated in Figure 13. This figure shows a graphical representation of the architectural description of a compiler. The components are the lexer, syntactic analyser, semantic analyser, optimiser, coder, symbols table and parse tree. These last two are data stores, while the first five represent computational elements. The connectors are represented by arrows and express data fetch or store in the repositories or data flow between the computational elements. The architectural configurations are implicit in the drawing, and they express which connector is related with which component.

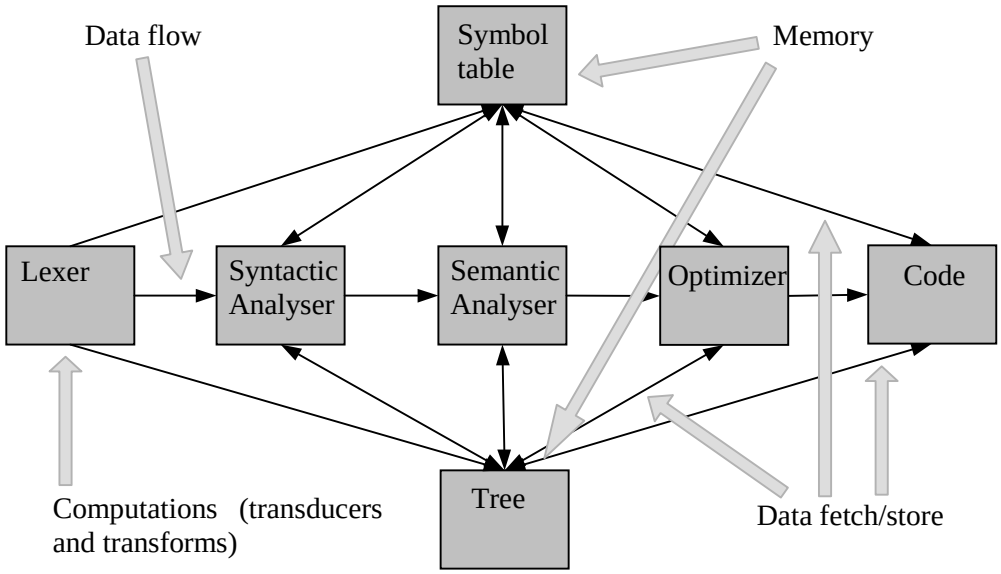

Figure 13. Compiler architecture [Garlan&Shaw96]

In contrast to Garlan and Shaw, Perry and Wolf [Perry&Wolf92] define a software architecture as

Software Architecture = {Elements, Form, Rationale}

They distinguish among *processing elements*, *data elements* and *connecting elements*. Processing and data elements would correspond to components and connecting elements to connectors. The *form* consists of weighted properties and relationships. The properties are used to constrain the choice of architectural elements and relationships are used to constrain the placement of architectural elements, i.e. the interaction with others. The weight is given by the importance of the property or relationship, allowing to select between different alternatives and to distinguish 'decorative' aspects. The *rationale* expresses the motivation for the choice of elements, and the form. It is represents the 'underlying philosophical aesthetics that motivate the architect'. In comparison with the previous definition, *elements* would correspond to connectors and components, the *form* would correspond to the configuration and the *rationale* is not present in the previous definition.

3.1.3 Architectural styles

If we consider an architecture as a formal arrangement of architectural elements, then an architectural style 'abstracts elements and formal aspects from various specific architectures' [Perry&Wolf92]. An architectural style is not as complete as a specific architecture, since the architecture contains more details on the particular system that it is modelling. The important thing about styles is that they encapsulate important decisions about the architectural elements and they relationships [Perry&Wolf92].

Summarising, an architectural style is a 'set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done' [Shaw&Clements96].

Consider clients, servers, filters and layers as examples of types of components, and procedure call, event broadcast, database protocols and pipes as examples of types of

connectors. Then a style can be expressed in terms of a *vocabulary* of components and connector types, and *constraints* on how they can be combined [Garlan&Shaw96]. Examples of architectural styles are pipe-and-filter architectures, client-server architectures, and layered architectures. The pipe-and-filter style, for example, consists of filter components that read data from input streams and produce data on output streams, and pipes, which bind the output stream of one filter to the input stream of another. The pipes-and-filter style emphasises sequential transformation of data [Oreizy&Taylor98].

3.2 Architecture Description Languages

Researchers found that in order to obtain the benefits of the architectural focus, software architectures must be provided with specification languages and analysis techniques [Medvidovic&Rosenblum97]. In this section we will describe what an architecture description language is and which are the desired characteristics an ideal ADL should exhibit.

An *architecture description language* (ADL) is a language that provides features for modelling a software system's conceptual architecture. They provide a concrete syntax and a conceptual framework for characterising architectures [Medvidovic&Taylor97].

An ADL's conceptual framework often relies on a formal semantic theory. Examples of formal semantic theories are Petri nets, Statecharts, communicating sequential processes (CSP), etc. The underlying semantic theory influences the suitability of the language to model particular kinds of systems (for example highly concurrent systems) or particular aspects of a given system (for example its static properties) [Medvidovic&Rosenblum97]. These aspects of the system that the ADL focuses on, which are expressed in its conceptual framework, constitute what Medvidovic and Rosenblum call architectural domain of concern [Medvidovic&Rosenblum97]. In section 3.2.1 we will explore different domains of concern enumerated by Medvidovic and Rosenblum. The relationship between the architectural domain and the candidate formal notations is not straightforward.

Medvidovic and Taylor proposed a framework to compare and classify ADLs [Medvidovic&Taylor97]. They state that an ADL must provide the means to explicitly specify components, connectors and architectural configurations. It is essential that also *components' interfaces* are treated as an explicit feature in an ADL. Other features that are desirable are *types*, *semantics*, *constraints* and *evolution* for components and connectors, and *understandability*, *hetereogeneity*, *compositionality*, *constraints*, *refinement* and *traceability*, *scalability*, *evolution* and *dynamism* for configurations. The tools that accompany a certain ADL somehow define its functionality. There are also requirements for the tool support that an ADL provides, such as *active specification*, *multiple views*, *analysis*, *refinement*, *code generation* and *dynamism*.

The focus on conceptual architecture and explicit treatment of connectors as first class elements differentiate ADLs from module interconnection languages, programming languages and OO notations.

Next we will explore the different domains of concerns addressed by different ADLs. In the rest of the chapter we will focus on the concern of architectural evolution, which is the specific area this dissertation is about.

3.2.1 Domains of concern of ADLs

The different domains of concern of ADLs have been classified and defined by Medvidovic and Rosenblum [Medvidovic&Rosenblum97] in the following way:

1. **Representation:** The role of the representation of an architecture is to help in understanding and to facilitate the communication about a system to different persons. The descriptions written in some ADLs that are simpler to understand than thatn the descriptions written in others.
2. **Design Process Support:** In decomposing a system into smaller building blocks design techniques, methodologies and tools are needed and can be supported by an ADL.
3. **Analysis:** Given that architectures often model large distributed or real-time systems abstracting only the issues that are relevant to the overall system organisation, sometimes it is easier to check for errors in the architecture than in a later phase. The different analysis an ADL supports is pretty much dependant on the semantic model that supports it. There are two kinds of analysis:
   3.1. **Static Analysis,** which consists of internal consistency checks such as determining whether the interfaces of connected elements match, whether constraints are satisfied, whether the semantics of the combination of the elements is the expected one, etc, and also some properties of concurrent systems such as absence of deadlock, starvation, etc.
   3.2. **Dynamic Analysis,** such as testing, debugging, assertion checking, assessment of the performance, reliability, etc.
4. **Evolution:** Evolution can occur at specification time or at execution time. We will not go into detail in the description of this concern since it will be covered in the following section.
5. **Refinement**: The specification of architectural models at several levels of abstraction is needed. A high level specification can be used for communication and understanding, while a lower level one can be used to analyse consistency of interconnections, and an even lower level one can serve as the basis for simulation. Code generation can be considered the last level of specification. This justifies the need of support for consistent refinements.
6. **Traceability:** Both traceability among the changes that occur in the different views of a system, and traceability of the requirements are needed. ADLs only provide support for traceability between the graphical and the textual views.
7. **Simulation/Executability:** Some dynamic properties, like reliability, require a running system in order to be assessed. Prototypes can provide information that cannot be obtained with a static description. The automatic generation of this system is the ultimate goal of some of the ADLs that address this concern.

Different ADLs provide support for different concerns. In our case, the concern we address is evolution.

3.3 Evolution of software architectures

Software evolution requires support that includes techniques and tools to aid interchange, reconfiguration, extension and scaling of software modules or systems[Medvidovic&al99]. It

is necessary to develop support that helps in determining what to change, facilitate the reasoning about the consequences of a change and govern changes so that the integrity of the system is preserved [Oreizy98]. Software architectures are a good starting point to provide a foundation for systematic software evolution, since they take into account the overall structure of the system and help in abstracting away from unnecessary details [Oreizy98][Oreizy&Taylor98]. Being the architecture a high level view of the system, architectural changes have to be carefully handled since the changes proposed at the architectural level will have consequences in many software assets.

In this section we will provide some insight in the state of the art concerning evolution of software architectures. There are two kinds of evolutions: those that occur while the system is running (*execution-time evolution*) and those that occur at specification-time (*specification-time evolution*). Even if this dissertation mainly addresses specification-time evolution, in this section we will describe both of them in order to provide a broader spectrum of the area and found the basis for the later analysis of different projections of our own approach.

3.3.1 Execution-time evolution

'Continuous availability is a critical requirement for an important class of software systems. For those systems runtime system evolution can mitigate the costs and risks associated with shutting down and restarting a system for an update.' [Oreizy&al98] Not only safety-intensive, mission-critical systems, such as systems like air-traffic control, telephone switching and high-availability public information systems but also commercial software applications [Oreizy&al98] require dynamic reconfiguration. Runtime extensions are available in some popular operating systems (e.g. dynamic links libraries in Unix) and components object models (e.g. CORBA and COM) by allowing new components to be located, loaded and executed during runtime. However they do not ensure consistency, correctness or system integrity during the course of the change or later [Oreizy&Taylor98]. These are the reasons that justify that a lot of researching has been focused on runtime evolution of software architectures.

According to Oreizy, purely syntactical models do not capture enough information of the application to ensure system integrity during the application of a change. Consequently all the ADLs that focus on dynamic evolution have incorporated some way of specifying dynamic behaviour [Oreizy98]. Moreover, Oreizy and Taylor state that in order to enable dynamic architectural descriptions, apart from an ADL, two other descriptions are needed:

- Architecture modification languages (AML) to describe the modification operations needed to modify the architecture. These languages are likely to be operational.
- Architecture constraint languages (ACL) to describe modification constraints. These languages describe the constraints under which the modification can be performed. These languages are likely to be declarative.

Dynamic changes of an architecture may be *planned* (*constrained dynamism*) or *unplanned* (*pure dynamism*). In planned dynamism all the changes have to be known a priori and are specified as part of the architectural model [Medvidovic&Rosenblum97]. On the other hand, systems that support unplanned changes have no restrictions on the kind of allowed changes: the ADL provides a feature that allows specifying changes while the system is running. Both planned and unplanned changes must be constrained and handled in such a way that there is no violation of the architectural properties.

The ADLs Rapide [Luckham&Vera95] and Darwin [Magee&Kramer96] support constrained dynamism. Also the formalism called Chemical Abstract Machine (CHAM) [Wermelinger98]that aims at specifying the computational behaviour of software architectures supports constrained dynamism. The chemical model views computation as a sequence of reactions between data elements, called molecules. The possible reactions are given by rules that specify the changes in the molecules. Components, connectors and data are modelled by means of molecules. Wermelinger proposes to model architecture style and reconfiguration by two different CHAMs, one for the creation and another for the evolution of the modelled architecture [Wermelinger98][Wermelinger98][Wermelinger98][Wermelinger98][Wermelinger98][Wermelinger98][Wermelinger98]. According to Oreizy, this formalism 'elegantly unifies an application's behavioural model and the rules governing runtime changes' but fails in capturing the conceptual model used by system designers [Oreizy98].

Darwin [Magee&Kramer96] and C2 [Oreizy&Taylor98][Oreizy98][Oreizy96][Oreizy&al98] support 'pure dynamism'. Darwin supports deletion and rebinding of components by interpreting Darwin scripts, and supports analysis of liveness and safety properties during the course of a runtime change by means of a behavioural model based on finite state machines.

C2 is an architectural style that facilitates runtime reconfiguration [Oreizy&Taylor98]. There is a Java framework that helps in the development of applications in Java style [Oreizy&al98]. ArchStudio is a prototype tool suite to support runtime reconfiguration applications built with this framework. Figure 14 depicts a high level view of the ArchStudio tool.

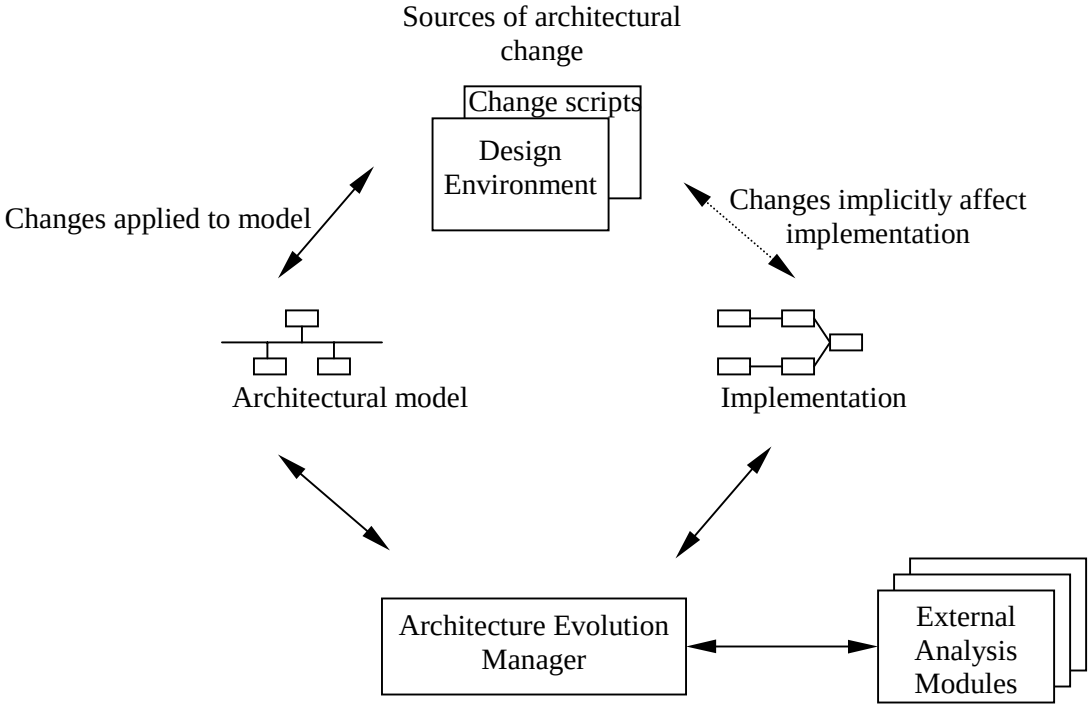

Figure 14. ArchStudio

In this figure, the architectural model is an up-to-date model of the application's architecture. It represents connections between components and connectors and their mapping to Java classes. A change request can be generated with some specific tools, some of them graphical and some of them textual, and they allow loading new components, linking them, deleting existent components, etc. The Architecture Evolution Manager (AEM) maintains the correspondence between the model and the implementation. When a change is requested, the AEM determines whether or not the invocation is valid. This task is done with the help of External Analysis Modules, which specify constraints on the modifications. But the operation of these External Analysis Modules is not described.

In his section we have presented an overview on current research in the area of runtime evolution of software architectures. Most of the works in this area only address planned evolution. These approaches are very restrictive because all the changes have to be foreseen in advance, when the architecture is designed. On the other side, the work of Oreizy and others supports unplanned evolution but only for architectures developed under the C2 style, built using the framework they provide. Moreover, they rely on the existence of External Analysis Modules to determine if the change request is safe, but do not specify which kind of analysis these modules do.

3.3.2 Specification-time evolution

The concerns in specification-time evolution of architectures are different from the ones at execution time. First of all, handling for unexpected changes is imperative. Second, one of the goals of the approaches that focus on specification-time evolution is to provide support for reuse of architectural specifications, and this aspect is not a concern in runtime evolution. Not many ADLs provide support for specification-time evolution, and most of the ones that do rely on the type system of the chosen implementation language.

Evolution can affect the entire architecture of a system or simple components and connectors in it [Medvidovic&Rosenblum97]. If we consider the components and connectors as types that are instantiated each time they are used, then we can view their evolution as subtyping. Different ADLs provide subtyping mechanisms. Aesop [Garlan&al94] enables behaviour-preserving subtyping of components and connectors to create substyles of a given architectural style. Rapide's [Luckham&Vera95] interface types can inherit from other types in an OO approach. ACME supports structural subtyping. C2 [Medvidovic&al99] allows multiple subtyping relationships among components: name, interface, behaviour and implementation subtyping, as well as combinations of them.

On the level of architectures, changes can reflect the evolution of a single system (what we will call proper evolution), or a system that evolves into a family of related systems (what we call reuse). Both mechanisms to scale architectural descriptions and to express incomplete architectural specifications are needed. Concerning the scaling of architectures, in general those systems that model configurations explicitly are better suited for incremental modifications of a system since the addition of components and connections is modular and thus easier [Medvidovic&Rosenblum97].

The possibility of specifying incomplete descriptions and handling them is important mainly in the design phase, when the details are not established yet and some decisions are deferred. However most of the systems have been built to forbid incomplete specifications [Medvidovic&Rosenblum97].

The aspect that almost has no support in the existing ADLs is application families. As the number of components grow linearly, the number of architectures that can be made out of them grow exponentially [Medvidovic&Rosenblum97], so organisation and versioning of related architectures is imperative. ACME provides a language construct to specify the family to which the architecture belongs, but more support is needed.

Support for application families is essential in software development environments based on product-line architectures. In these environments there is a distinguished architecture, a template, that serves as the basis for defining a set (family) of related applications. Each time a new customer arrives with a certain set or requirements the new system is built based on the template architecture, possibly making small changes to it. The following kind of situations can arise: the template-architecture, the one that forms the basis of the product line, is evolved in order to improve performance. At the same time a customisation based on the (old) template architecture is being developed, to address the requirements of a new client. Can the new template be integrated in the new customised version of the architecture without problems? Will the effect of this integration be the expected one? Can the new template be integrated in previous customisations of the architecture? The situation is illustrated in Figure 15. No ADL provides support for these situations.

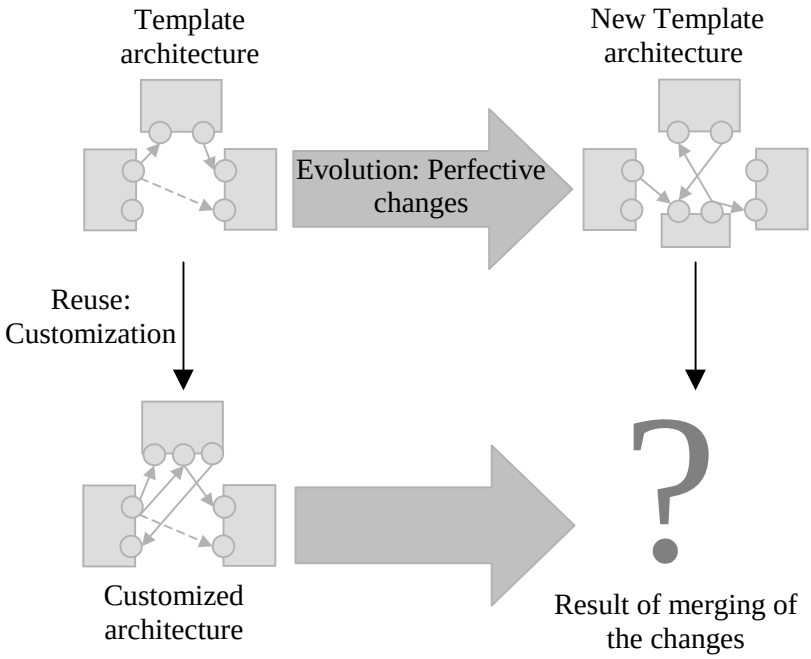

Figure 15. Simultaneous evolution

3.4 This dissertation

In this section we will briefly discuss the particular scope of this dissertation in the context of the research work in the area of software architectures.

The problem of simultaneous evolution, which was introduced in section 3.3.2 as appearing in environments based on product-line architectures, can be found in any other collaborative

environment as well. Given two different evolutions for a given architecture, the problem consists in determining if the merge of the two will produce the expected result, or their interaction can cause unexpected problems. To our knowledge, there is no previous work in this particular topic. This is the problem we address in this dissertation.

Given a tool that can determine if the interaction between different independent evolutions generates conflicts, it could be integrated in the Change Manager of an architecture evolution tool. For example, it could integrate the external modules in ArchStudio, the tool suite described in section 3.3.1.

We propose a language called Adela. Adela has constructs that enable the modification of an architectural description. However, Adela it is not an architecture modification language (defined in section 3.3.1) but an architectural description language. Adela's underlying semantic theory relies on the reuse contract formalism presented in chapter 2.

In order to be able to concentrate our effort in the effect of simultaneous evolutions, particular problems of runtime evolution will not be addressed. We will only focus on specification-time evolution. However, we think that the approach can be extended in order to address runtime evolution as well.

Adela is a very simple language, whose concern is limited to evolution. It does not intend to be a general-purpose architecture description language.

Other feature explored in this dissertation, based on the same approach (i.e. reuse contracts) is reuse of architectural descriptions. Support for reuse is provided by means of the definition of styles and by means of incremental definition of architectures. A base architectural description not completely specified can act as a specification for other more specific and refined descriptions, which adapt the original description for a particular need. The original specification determines in this way a family of related architectural descriptions, which is composed by all the descriptions that adapt it. In order to do this it is necessary to have means of expressing incomplete specifications and verifying that a more complete description complies with another one.

3.5 Summary

In this chapter we have presented the main concepts, problems, ideas, needs and research directions in the area of architectural evolution. We have seen that in the area of evolution a big amount of work is devoted to runtime changes. Even in specification-time changes can be seen as special case of dynamic changes, some of their concerns are different. While runtime evolution techniques often use complicated formalisms to be able to ensure that the system will go on running during the course of a change, specification-time evolution approaches deal not only with incremental evolution, but also with reuse of architectural descriptions. However, there is a big area that has not been explored yet, which is related to the support and detection of conflicts in the simultaneous evolution of architectural descriptions. A solution for this problem is the main goal of this dissertation.

4 ADELA: A SIMPLE LANGUAGE TO DESCRIBE SOFTWARE ARCHITECTURES

In this chapter the architecture description language Adela is introduced. First, we will explain what are the main features provided by Adela in order to describe an architecture. We will later present the syntax of Adela, and how is it related to the reuse contracts model. As an example we will describe the architecture of the system SOUL in Adela. SOUL is an interpreter of a declarative language embedded in Smalltalk. This example will also be our starting point to speak about evolution of software architectures in later chapters. The chapter finishes with some comments on how Adela was implemented.

4.1 Some characteristics

Adela is a very simple language to describe software architectures. Adela's language constructs were chosen so that they are as simple as possible, in order to make it easier to find the conflicts their interactions can cause. However this minimalistic language with few constructs can serve as the basis to build other more complex languages. It is meant to be extended with more specific or complex constructs whenever they are needed. Successive refinements of this language will enrich the original conceptual framework supported by Adela with more sophisticated features. Many refinements can be imagined and some of them are analysed in Chapter 5. The need to have very simple language constructs forced us to define a new ADL instead of using an existent one.

As this language is specially developed to deal with evolution of software architectures, an architectural description expressed in this language does not explicitly describe the architecture itself, but it describes it implicitly by enumerating the operations that have to be performed in order to obtain this architecture. However this is not an architecture modification language, but an architecture description language, since an expression in Adela incrementally describes an architecture.

The underlying semantic theory of Adela relies on the reuse contracts formalism presented in Chapter 2. The conceptual framework Adela supports is the subject of the next section.

4.2 Architectural conceptual framework

Recall from section 3.1.2 that Medvidovic and Taylor propose that a conceptual architecture should have as building blocks components, connectors and architectural configurations (section 3.2). Allen and Garlan discuss the necessity of distinguishing between connectors and components. The advantage of representing connectors and components using the same concept is that the language is simpler as well as the definition of its semantics. But if the

purpose of the language is to provide a 'vehicle of expression that matches the intuition and practices of users, then components and connectors are to be distinguished [Allen&Garlan97].

Based on this analysis, and based on the fact that Adela does not focus on any specific concern but evolution, we will not differentiate connectors and components. They will all be represented as *architectural elements*.

So, in Adela, each *architecture* has a name and describes

- a set of *elements*, and
- their *interconnections* (the architectural configuration).

Each *element* has a name, a set of gates and possibly a description of the internal workings of the element.

Gates are the interaction points in the elements, the set of gates in an element may be seen as its interface. For example, if the element represents a server, its gates may represent the services provided by the server. The internal workings of the element are described by an *internal description*, and a set of *bindings*. The *internal description* of an element is defined in terms of another architecture. Each element can have at most one such description. The *bindings* are mappings from *gates in the described element to gates in the elements that take part in the architecture that describes it*. The bindings define the interaction of the describing elements to the outside world. They connect the internal description with the external interface of an element.

Bindings have a direction that indicates that there is information or control flowing from the source binding to the target binding. There cannot be parallel bindings, i.e. bindings that communicate the same pair of gates, unless they go in the opposite direction, meaning that the flow is bidirectional. Bindings provide the only means of interaction among elements belonging to different architectures.

The internal description together with the bindings provide a powerful abstraction mechanism. The internal description of an element is hidden inside it, and can only access and communicate with entities at a higher level of abstraction through the gates that are defined in the described element. The relationship between an element and its internal representation can be seen as a refinement.

The interconnections among the elements inside a given architecture are represented by *links* that map gates to other gates. Links relate gates that belong to elements defined inside the same architecture. Only one link is allowed from a certain gate to another, expressing that there is some flow of information or control from the source gate to the target gate. Like in the case of bindings, parallel links are only allowed if they go to the opposite direction.

In Figure 16 the relationship of the different architectural constructs is illustrated. The figure depicts an architecture that has two interacting elements. One of the elements is further refined in terms of another architecture.

The square entities in the figure are elements. The oval ones are architectures. Since there is at most one architecture per element, we will omit drawing the ovals in the following figures. This does not mean that the architectures are not needed as a building block, they are a very important abstraction. The small circles are gates. The lines that relate them are links or bindings.

We call an *architectural description*, or a *model*, a set of architectures that describe a system.

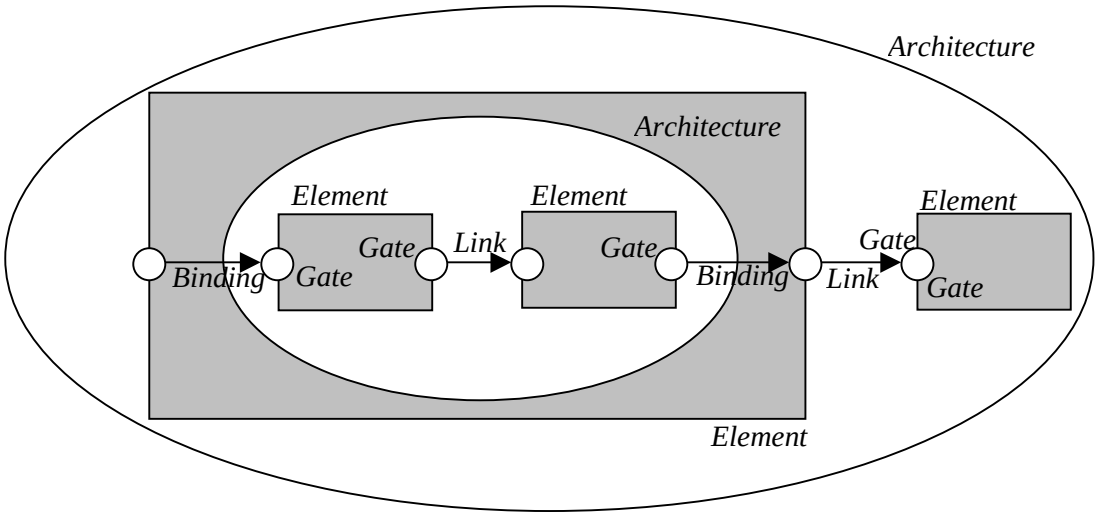

Figure 16. Elements in an architectural description

4.2.1 Example: SOUL's architectural description

SOUL is a PROLOG-like declarative rule-based language implemented in Smalltalk [Wuyts99]. It allows the declaration of structural rules about Smalltalk source code. These rules can be used to query the software system to find occurrences of certain structures or to enforce the presence of structures [Mens&Wuyts99].

SOUL's architecture is depicted in Figure 17. The element query-interpreter receives a query and produces a result by interacting with the repository and by making use of the working memory.

The element called reify1 receives an element in its gate as-data and returns the representation of this element by its gate as-object. In this particular case, it gives a representation of the query to the query interpreter. The query interpreter takes the representation of the query as the goal to interpret. This goal is interpreted by means repeated application of unification with the rules and facts in the repository and decomposition into subgoals.

The rules and facts in the repository are looked up by means of the rule selector element. The rule selector element receives the term to match, transforms it into a key and uses the key to access the repository. The repository returns the clauses that match with the term, and the result is then returned to the query interpreter.

When the query interpreter receives the clauses that match with the goal, it unifies them wit the goal generating bindings. All the intermediate bindings that are assigned to variables during unification are stored in the working memory. The query interpreter can add a new binding to the working memory, or read all the bindings available in the memory. The element DP2 represents data passing. The element reify2 transforms the working memory into its representation so that the query interpreter can operate with it.

If the goal of one of the subgoals in which the initial goal is decomposed requires it, some new facts and rules can be asserted to and/or retracted from the repository.

When the whole process finishes, and there are no more goals to match, the resulting bindings are returned as a result.

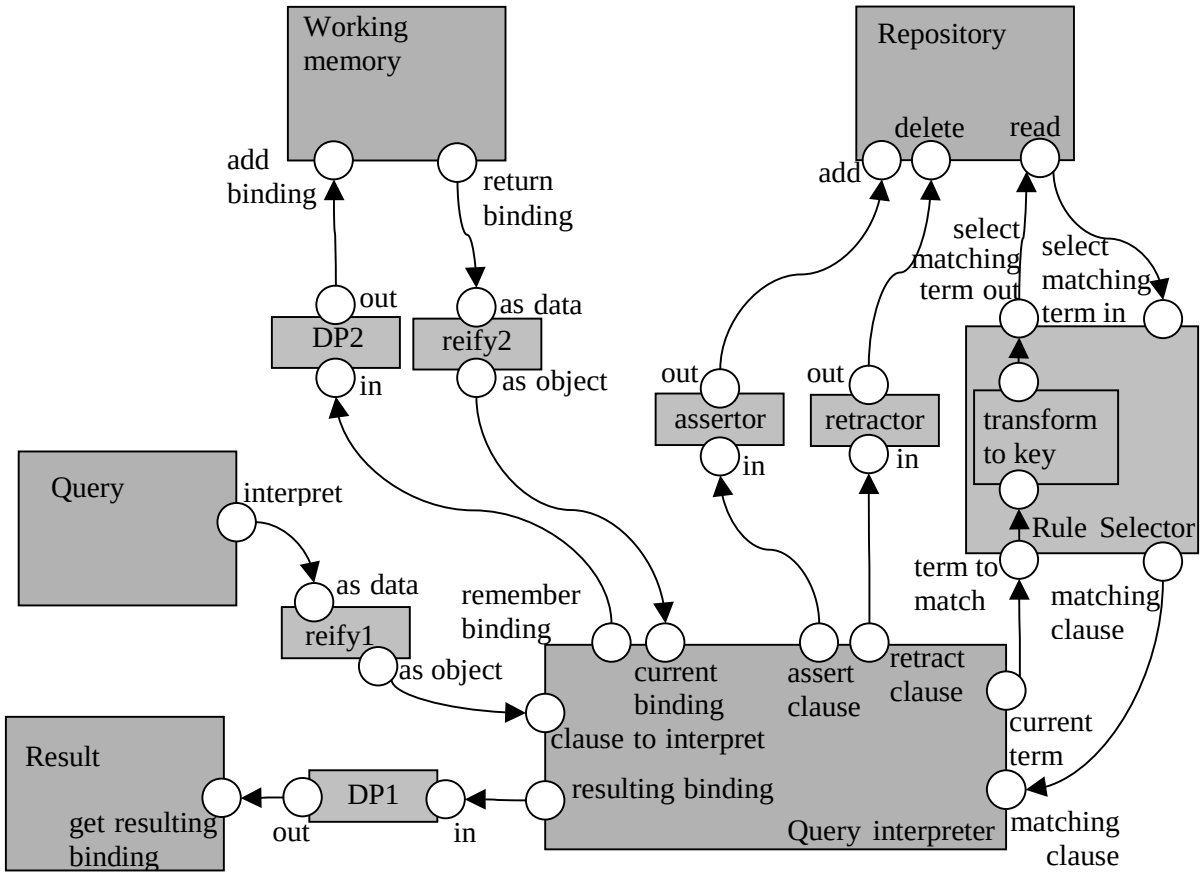

Figure 17. The architecture of SOUL

In this very simple architectural description we can already see that some elements are more concerned with the communication (such as reify, DP, assertor, retractor, and even the rule selector), others are computational entities, like the query interpreter, and others are data stores, such as the repository or the working memory.

Many architectural descriptions can be imagined for the same system. This one shows the flow of information between the different elements. Another view could show us the static structure of the system in terms of the classes that take part in it. Every high level view of the system illustrates a different aspect.

Even if it does not explicitly expresses its semantics, the architectural description gives us an idea of the way SOUL operates.

4.3 Syntax

A description in language Adela, called AdelaDescription, describes an architectural model, i.e. a set of architectures. This is done incrementally. The starting point is an empty architectural description, i.e. one without any architecture. This initially empty architectural description is modified by *increments* (also called operations).

AdelaDescription ::= Name**(**ModelDefinition**)**[2]

ModelDefinition ::= Increment | ModelDefinition**,** Increment

Increment ::= BasicIncrement | ParallelIncrement

There are basic increments and parallel increments. Basic increments describe an evolution of the architecture. Parallel increments describe several evolutions that are performed in parallel by independent modifiers. When a parallel increment is applied to the architectural model special attention is required with respect to the possible unexpected interactions between the different modifications.

ParallelIncrement ::= **parallel(**BasicIncrement**,** BasicIncrement**)**

BasicIncrement ::= Modification Type**(**Modifier**)**

Note that only two parallel increments are allowed. We initially defined it in this way because this makes it easier to implement and to map onto the reuse contracts model, but it does restrict the expressiveness of the language somewhat.

Basic increments are represented using an *increment type* and a *modifier*. The increment type expresses the operation we want to perform, and the modifier determines the parameters. For different increment types different modifiers are needed.

IncrementType(Modifier) ::=

addArchitecture(Arch) |

removeArchitecture(Arch) |

addElement(Arch,Elem) |

removeElement(Arch,Elem) |

moveElement(Arch,Elem,HostArch) |

implementElement(Arch,Elem,DescrArch) |

abstractElement(Arch,Elem, DescrArch) |

addGate(Arch,Elem,Gate) |

removeGate(Arch,Elem,Gate) |

addLink(Arch,SElem,SGate, TElem,TGate) |

removeLink(Arch,SElem,SGate, TElem,TGate) |

addBinding(Arch,SElem,SGate, TArch,TElem,TGate) |

removeBinding(Arch,SElem,SGate, TArch,TElem,TGate)

[2] The ones in bold are terminal symbols.

Arch, DescrArch, Tarch, HostArch are names of architectures.

Elem, SElem, TElem are names of elements.

Gate, Sgate, TGate are names of gates.

4.3.1 Naming and scoping

We will use some scoping conventions that make the architectures more readable and easy to understand and implement. Names of architectures are unique inside the model. Elements are uniquely identifiable by their name inside an architecture so there is at most one element with a certain name inside an architecture. Gates are uniquely identifiable by their name inside an element. Thus, in order to address an architecture inside a model it is sufficient to provide its name. In order to address an element it is necessary to give both the name of the element and the name of the architecture in which it is defined. And to address a gate three names have to be provided, i.e. architecture, element and gate.

This naming conventions are the reason why the modifiers of some increment types are so complex in some cases (e.g. addLink(A,SE,SG, TE,TG)). An alternative solution would be to use unique names for every entity, but we found that this option limits the abstraction capabilities and very soon you run out of meaningful names. A third option would be to make compound names for gates and elements, so that the name of the architecture is part of the name of the elements defined in it.

Notice that even when there is nesting of architectures, no matter the level of nesting, the name of architectures are considered unique. This has the effect of avoiding arbitrary long paths in order to address a certain entity.

4.3.2 Example: SOUL's architectural description

The expression in Adela to denote SOUL's architecture, the architecture depicted in Figure 17, is:

```
mySoulModel(soul([
        addArchitecture(soul_architecture),
% ELEMENTS[3]
% query
        addElement(soul_architecture,query),
        addGate(soul_architecture,query,interpret),
% result
        addElement(soul_architecture, result),
        addGate(soul_architecture,result,get_resulting_bindings),
% working_memory
        addElement(soul_architecture,working_memory),
        addGate(soul_architecture,working_memory,add),
        addGate(soul_architecture,working_memory,return),
% query_interpreter
        addElement(soul_architecture,query_interpreter),
        addGate(soul_architecture,query_interpreter,goal_to_interpret),
        addGate(soul_architecture,query_interpreter,resulting_bindings),
```

[3] This is an Adela description the prototype interpreter can interpret. The lines that begin with % are comments.

```
        addGate(soul_architecture,query_interpreter,remember_binding),
        addGate(soul_architecture,query_interpreter,matching_clauses),
        addGate(soul_architecture,query_interpreter,current_term),
        addGate(soul_architecture,query_interpreter,current_bindings),
        addGate(soul_architecture,query_interpreter,assert_rule),
        addGate(soul_architecture,query_interpreter,retract_rule),
% repository
        addElement(soul_architecture,repository),
        addGate(soul_architecture,repository,read),
        addGate(soul_architecture,repository,assert),
        addGate(soul_architecture,repository,retract),

% CONNECTING ELEMENTS
% rule_selector
        addElement(soul_architecture,rule_selector),
        addGate(soul_architecture,rule_selector,select_matching_terms_in),
        addGate(soul_architecture,rule_selector,select_matching_terms_out),
        addGate(soul_architecture,rule_selector,term_to_match),
        addGate(soul_architecture,rule_selector,clauses_that_match),
% reify_1
        addElement(soul_architecture,reify_1),
        addGate(soul_architecture,reify_1,reified_as_object),
        addGate(soul_architecture,reify_1,reified_as_data),
% reify_2
        addElement(soul_architecture,reify_2),
        addGate(soul_architecture,reify_2,reified_as_object),
        addGate(soul_architecture,reify_2,reified_as_data),
% dp_1
        addElement(soul_architecture,dp_1),
        addGate(soul_architecture,dp_1,in),
        addGate(soul_architecture,dp_1, out),
% dp_2
        addElement(soul_architecture,dp_2),
        addGate(soul_architecture,dp_2,in),
        addGate(soul_architecture,dp_2, out),
% assertor
        addElement(soul_architecture,assertor),
        addGate(soul_architecture,assertor,in),
        addGate(soul_architecture,assertor,out),
% retractor
        addElement(soul_architecture,retractor),
        addGate(soul_architecture,retractor, in),
        addGate(soul_architecture,retractor, out),

% ARCHITECTURAL CONFIGURATION
% reify_1
        addLink(soul_architecture,query,interpret,reify_1,as_object),
        addLink(soul_architecture, reify_1, as_data, interpreter, goal_to_interpret),
% dp_1
        addLink(soul_architecture,interpreter,resulting_bindings, dp_1,in),
        addLink(soul_architecture,dp_1,out,result,get_resulting_bindings),
% dp_2
        addLink(soul_architecture,interpreter,remember_binding,dp_2,in),
        addLink(soul_architecture,dp_2,out,working_memory,add),
% reify_2
```

```
            addLink(soul_architecture,working_memory,return, reified_2,as_data),
            addLink(soul_architecture,reified_2,as_data, interpreter,current_bindings),
    % assertor
            addLink(soul_architecture,interpreter,assert_rule, assertor,in),
            addLink(soul_architecture,assertor,out, repository,assert),
    % retractor
            addLink(soul_architecture,interpreter,retract_rule, retractor,in),
            addLink(soul_architecture,retractor,out, repository,retract),
    % rule_selector
            addLink(soul_architecture,rule_selector,select_matching_terms_out,
            repository,send),
            addLink(soul_architecture,repository,send,
            rule_selector,select_matching_terms_in,),
            addLink(soul_architecture,interpreter,current_term, rule_selector,
            term_to_match),
            addLink(soul_architecture,rule_selector,clauses_that_match, interpreter,
            matching_clauses),
    % INTERNAL DESCRIPTION OF RULE SELECTOR
            addArchitecture(rule_selector_arch),
            implementElement(soul_architecture,rule_selector, rule_selector_arch),
    % transform_to_key element
            addElement(rule_selector_arch,transform_to_key),
            addGate(rule_selector_arch, transform_to_key,term_to_match),
            addGate(rule_selector_arch, transform_to_key,select_matching_terms),
    % Bindings
            addBinding(soul_architecture,rule_selector,term_to_match,
            rule_selector_arch,transform_to_key,term_to_match),
            addBinding(rule_selector_arch,transform_to_key,select_matching_terms,
            soul_architecture,rule_selector,select_matching_terms_out)
      ].
```

None of the increments removeArchitecture, removeGate, removeLink, removeElement, removeBinding, abstractElement and moveElement were used in this description. The reason is that those increments are more related to evolution than to the description of an architecture, since they imply modifications to existing elements.

It is also important to notice that the language is not compact at all. In fact, the concern in Adela is not to make a compact language, but to make a language with very basic language constructs, so that we can analyse their simple behaviour with respect to each other. More complex increments, which might make the above descriptions shorter are analysed later in Chapter 6.

4.4 Semantics

In this section we will describe the semantics of an expression in Adela. We will do this in three parts. In the first part (section 4.4.1) the informal semantics of the increments is explained in terms of the changes they produce on the architectural model. In the second part (section 4.4.2), we define the desired semantics of an entire Adela expression. The purpose of this section is to provide a kind of specification for Adela's semantics. We illustrate this with an example. Finally, in section 4.4.4 we describe how do we define Adela in terms of the reuse contracts formalism.

4.4.1 Informal semantics of the increments

In this section we informally describe the semantics of the increments in terms of the modifications they produce in the architectural model.

The increments addArchitecture(A), addElement(A,E) and addGate(A,E,G) allow to introduce architectures, elements and gates in the model, similarly removeArchitecture(A), removeElement(A,E) and removeGate(A,E,G) eliminate existing architectures, elements and gates. There are some restrictions on the application of each of these increments, e.g. an architecture can only be introduced if there is no other architecture with the same name in the model, and an architecture can be removed from a model only if it does not have any gates or elements nested in it.

The increment moveElement(AE,E,A) moves an element with its gates and internal description from architecture AE to architecture A. Even though the same effect can be obtained by deleting the element and gates one by one, from architecture AE and introducing them again in A, we found this increment interesting because of two reasons. First, it allows doing the same in a simpler and more compact way and second, it captures more information about the intention of the modifier, enabling the detection of more meaningful conflicts. An example of such a conflict is described in section 5.3.

The increment implementElement(AE,E,A) attaches to the element E in architecture AE the internal description defined by architecture A. This operation can only be performed if the element E in AE does not have an internal description already. The internal description can later be disconnected from the element it describes by abstractElement(AE,E, A). After the application of this operation, architecture A is still in the model, but it does not describe element E any more. abstractElement(AE,E, A) can only be applied if the elements in architecture A are not connected by any binding with the element E in AE.

A new link can be introduced from a source gate SG in a element SE of the same architecture A to a target gate TG in an element TE in the same architecture A by means of addLink(A,SE,SG, TE,TG). This increment can be applied only if there is no other link from gate SG in SE in A to gate TG in TE in A, however, a parallel link in the opposite direction, i.e. from TG in TE to SG in SE is allowed. The link that goes from SE to TE can be removed by applying removeLink(A,SE,SG, TE,TG).

The increment addBinding(A,SE,SG, TA,TE,TG) introduces a new binding from gate SG in element SE in architecture A to the gate TG in element TE in architecture TA, which is the internal description of the element SE. It can be removed by removeBinding(A,SE,SG, TA,TE,TG). The restriction about parallel bindings is similar to the one in links.

4.4.2 Desired semantics

In this section the desired semantics of an Adela description is presented. The semantics of an AdelaDescription is a pair that contains the name of the architectural description, and the semantics of the model definition.

[] is a function that maps each term onto its semantics,

$[AdelaDescription] \in AdelaSemantics$

$AdelaSemantics = (Name \times ModelDefinitionSemantics)$

$[Name(ModelDefinition)] = (Name, [ModelDefinition])$ where

$$[ModelDefinition] \in ModelDefinitionSemantics$$

The semantics of the model definition have two components: the result from applying all the increments to the empty architectural model and a set of conflicts that appeared during these applications.

$$ModelDefinitionSemantics = (ModelSemantics \times Conflicts)$$

The semantics of a model is a set of semantics for the architectures,

$$ModelSemantics \in (P(ArchitecturalSemantics))$$

where each architecture semantics consists of a name, a set of elements and an architectural configuration.

$$ArchitecturalSemantics = ArchitectureName \times P(Element) \times Configuration$$

As we said before, each element has a set of gates, an internal description that is another architecture and a set of bindings. Each gate has a name. Bindings are mappings from gates to gates.

$$Element = ElementName \times P(Gates) \times InternalDescription \times P(Binding)$$

$$Gates = GateName$$

$$InternalDescription = ArchitectureName$$

$$Binding = GateName \times GateName$$

The architectural configuration is also a mapping from gates to gates.

$$Configuration = GateName \times GateName$$

$$ArchitectureName \subset Name,\ ElementName \subset Name,\ GateName \subset Name$$

The sets of names *ArchitectureName*, *ElementName* and *GateName* are pairwise disjoint.

The different conflicts we may find are the subject of discussion of a later chapter. So we will not define *Conflicts* yet.

Returning to the semantics of a model definition, recall that

$$[ModelDefintion] \in (ModelSemantics \times Conflicts)$$

and

ModelDefinition ::= Increment | ModelDefinition , Increment

The semantics of a model definition is defined incrementally by the function apply

$$[Increment] = apply\ Increment\ (\{\},\{\})$$

$$[Increment\ ,\ ModelDefinition] = apply\ Increment\ ([ModelDefinition])$$

and apply can be defined case by case for all the possible increments. For example, let's see what would it look like in the case of the basic increments addArchitecture and removeElement:

apply (AddArchitecture (ArchitectureName)) (M,C) =

((M ∪ {(ArchitectureName, {}, {}),C),

***if** everything is Ok for the application of addArchitecture*

(i.e. ArchitectureName is not the name of another architecture in the model)

((M,C ∪ {conflict on the application of addArchitecture}),

***if** addArchitecture cannot be performed*

When applying the increment AddArchitecture(ArchitectureName) to an architectural model we need to check whether the architecture with name ArchitectureName is already defined in that model. If so, we obtain a conflict that tells us that the operation is not possible. If not, we obtain a modified architectural model, with the new architecture.

apply(removeElement (ArchitectureName, ElementName)) (M, C) =

(M1, C) where M = M' ∪ {(ArchitectureName, E, C)}
E1 = E - {(ElementName, _, _, _)}
M1 = M' ∪ {(ArchitectureName, E1, C)}

***if** everything is OK for the application of removeElement*

(i.e. the ElementName is an element of ArchitectureName and it does not have gates nor implementation)

((M,C ∪ {conflict on the application of removeElement}),

***if** removeElement cannot be performed*

For the other increments we could give similar definitions. This defines the semantics we expect to obtain from an AdelaDescription. However, will use the semantic definition presented here as a specification because in fact we will use the reuse contracts formalism as a means of defining the semantics. The use of reuse contracts in the definition of Adela will be presented in section 4.4.4.

4.4.3 Example: SOUL's architectural description

As an illustration for the above semantic definitions, the semantics of the SOUL description in Adela is presented in this section. This semantics was generated by the prototype interpreter for Adela, with the Adela description presented in 4.3.2 as input.

empty represents the empty architecture, i.e. an architecture without nodes or edges.

```
soul_architecture([

  % ELEMENTS
  element(assertor,[gate(out),gate(in)],[],empty),
  element(retractor,[gate(out),gate(in)],[],empty),
  element(dp_2,[gate(out),gate(in)],[],empty),
  element(dp_1,[gate(out),gate(in)],[],empty),
  element(reify_2,[gate(as_data),gate(as_object)],[],empty),
  element(reify_1,[gate(as_data),gate(as_object)],[],empty),
  element(repository,[gate(retract),gate(assert),gate(read)],[],empty),
```

```
    element(query_interpreter,
        [gate(retract_rule),gate(assert_rule), gate(current_bindings), gate(current_term),
        gate(matching_clauses), gate(remember_binding), gate(resulting_bindings),
        gate(goal_to_interpret)], [], empty),
    element(working_memory,[gate(return),gate(add)],[],empty),
    element(result,[gate(get_resulting_bindings)],[],empty),
    element(query,[gate(interpret)],[],empty)],
    element(rule_selector,
        [gate(clauses_that_match), gate(term_to_match),
        gate(select_matching_terms_out), gate(select_matching_terms_in)],
        % BINDINGS
        [(rule_selector,term_to_match,transform_to_key,term_to_match),
        (transform_to_key,select_matching_term,rule_selector,
        select_matching_terms_out)],
        % INTERNAL DESCRIPTION
        rule_selector_arch(
        [element(transform_to_key,[gate(select_matching_term),gate(term_to_match)],
        [],empty)],
        [])),

    % CONFIGURATION
    [
    link(query_interpreter, retract_rule, retractor, in),
    link(query_interpreter,assert_rule,assertor,in),
    link(query_interpreter,remember_binding,dp_2,in),
    link(query_interpreter,resulting_bindings,dp1,in),
    link(query,interpret,reified _1, as_object),
    link(query_interpreter,current_term,rule_selector,term_to_match),
    link(working_memory,return_binding,reified_2, as_data),
    link(repository,read,rule_selector,select_matching_terms_in),
    link(retractor,out,repository,delete), link(assertor,out,repository,add),
    link(rule_selector,select_matching_terms_out,repository,read),
    link(reified _2,as_data,query_interpreter,current_binding),
    link(rule_selector,matching_clauses,query_interpreter,matching_clauses),
    link(reified _1, as_object,query_interpreter,clause_to_interpret),
    link(dp _2,out,working_memory,add_binding),
    link(dp _1,out, query_interpreter,get_resulting_bindings)
    ]
)

    % CONFLICTS
    C = []
```

The list of conflicts is empty because in this case it was possible to apply all the operations. Conflicts are the subject of Chapter 5.

4.4.4 Definition of Adela in term of reuse contracts

Now we have an idea of what the expected semantics of a term in Adela is. Defining the conflicts as in section 4.4.2 tends to give raise to fairly complicated semantics. Furthermore, it does not allow us to make use of earlier research on reuse contracts to do automated conflict detection. However, as we mentioned before, our approach will be to use reuse contracts as an underlying semantic theory. Following this approach we can consider that the reuse contract formalism will do something similar to what is specified in the definition of the apply

operation in section 4.4.2. The reuse contract formalism will be in charge of computing the actual semantics of the increments, as well as checking for the conflicts that appear when a certain condition for the application of an operation is not satisfied.

We will distinguish two conceptual levels: the level of software architectures and the level of the reuse contracts formalism. An AdelaDescription belongs to the level of software architectures. We will take an Adela description to the level of reuse contracts, the reuse contracts formalism will operate with it, performing the actual evolutions, and the result will be taken back to the level of software architectures.

The initial Adela description of the architectural model will be taken to the level of reuse contracts by translating each increment type into one or more contract types, and each modifier for the increment type into a modifier for that contract type. Subsection 4.4.4.2 describes to which contract type each increment corresponds. Once on the level of reuse contracts, the actual evolutions will be performed and the conflicts will be detected. In section 2.2 we explained that the reuse contracts framework operates on a labelled typed graph, and all the contract types express modifications on this graph. So the result of the application of the contract types will be a labelled graph and a set of conflicts. In order to take this result back to the level of architectural descriptions the graph will be interpreted as an architectural semantics and the conflicts will be translated into meaningful conflicts in the level of architectures. We do this by defining a two-way mapping between architectures and labelled typed graphs. In this way we will know whether a certain graph represents an architectural description, and which is the architectural description represented by a graph. This mapping is explained in subsection 4.4.4.2. The whole process is illustrated in Figure 18.

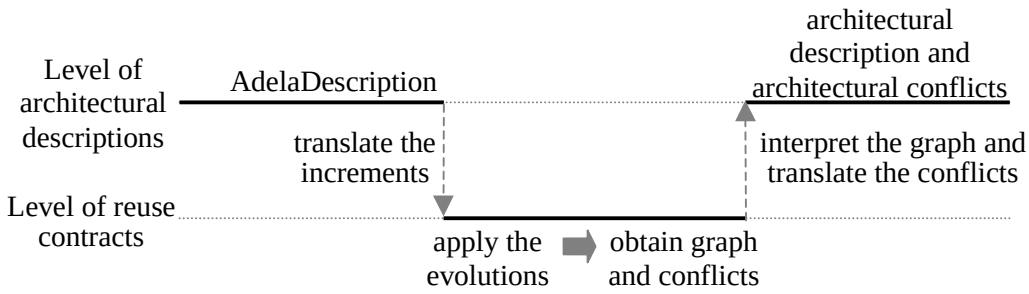

Figure 18. Adela and reuse contracts

4.4.4.1 Mapping from architectures to labelled typed graphs

We will define how can an architectural model be expressed in terms of a labelled typed graph, i.e. how are architectures, elements, gates, links and bindings expressed, and how are their relationships expressed.

We will represent architectures, elements, and gates as nodes in the graph, and all the other entities as edges. So there will be three types of nodes: architecture, element and binding. The names of the elements, gates and architectures will be represented by the labels. The fact that an element belongs to a certain architecture will be expressed by a nesting relationship, and likewise for the relationship between a gate and the element to which it belongs. Links will be represented by edges of type link. There will be also an edge type binding for bindings. The

relationship between an element and the architecture that defines its internal description will also be represented by a nesting. Figure 19 illustrates this mapping in a graphical way.

As we mentioned in section 2.2.2, each type can have constraints associated. These restrictions have to be satisfied by all the entities of that type, otherwise a conflict is raised. The restrictions often express restrictions on the types of nodes certain edges can connect. Table 2 shows the restrictions we defined for edges of types *nesting*, *binding* and *link*. The third column in the table indicates which conflict is raised in each case.

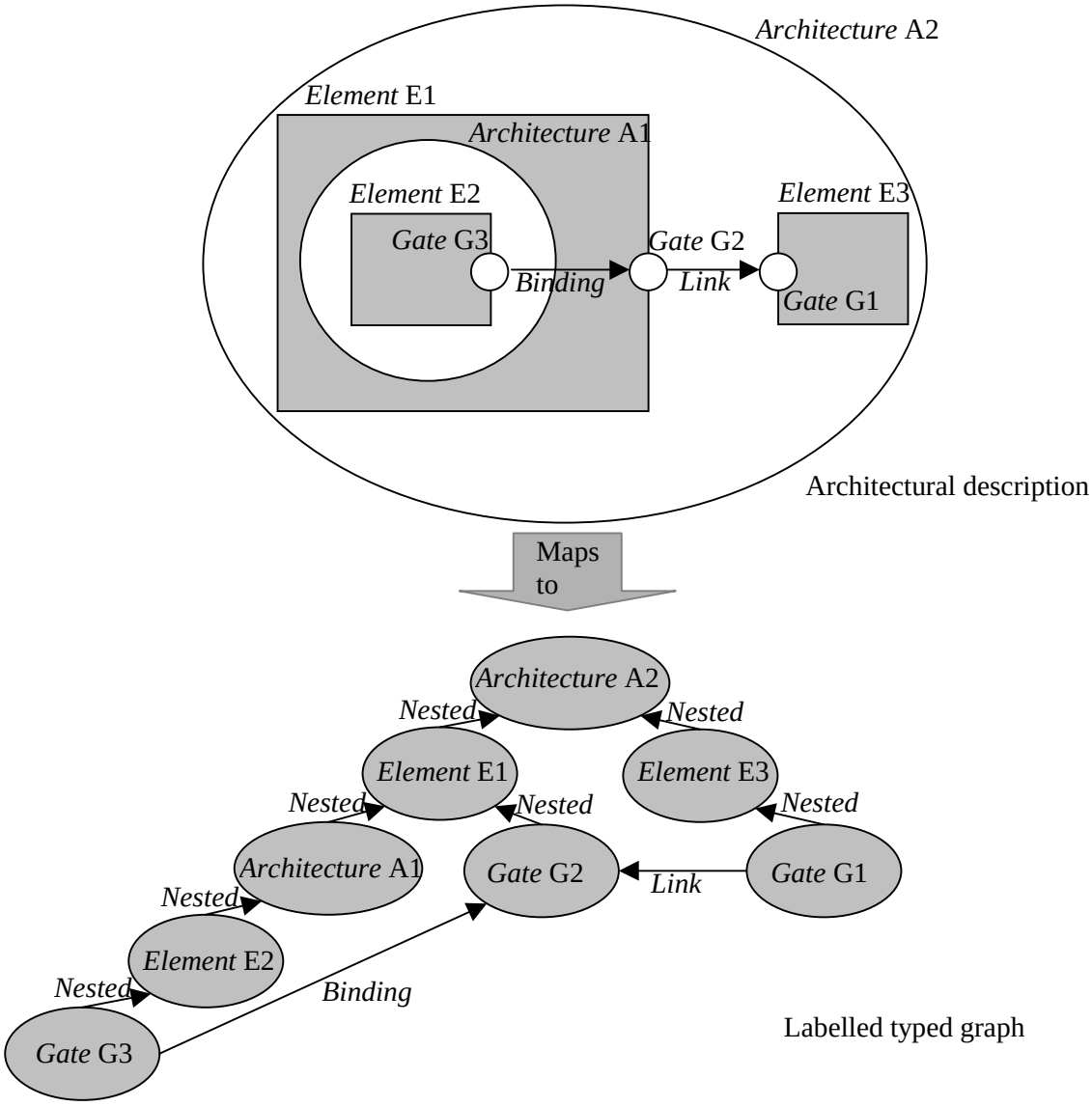

Figure 19. Mapping from architectures to labelled graphs

Table 2. Edge type constrains

Edge Type	Conditions	Conflict
Nested	A *nested* edge can relate a gate with an element, an element with an architecture and an architecture with an element.	Nesting Constraint
Bindings	A *binding* edge is only allowed between a gate in an element A and a gate of another element B only if A belongs to the internal description of B.	Binding Constraint
Link	A *link* edge can connect a gate with another if both gates belong to elements that belong to the same architecture	Link Constraint

4.4.4.2 Translation of increments into basic contracts types

The translation of Adela increments into the basic contract types in the reuse contract formalism is defined in Table 3.

Table 3. Translation of increments into contract types

Increment	Contract types
addArchitecture(a)	extension(a, *architecture*)
removeArchitecture(a)	cancellation,(a, *architecture*)
addElement(a,e)	nestedExtension(a, *architecture*, ename(a,e), *element*)
removeElement(a,e)	nestedCancellation (a, *architecture*, ename(a,e), *element*)
moveElement(a,e,ai)	redirectEdge(ae, ename(a,e), *architecture*, *element*, ai)
implementElement(a,e,ia)	refinement(”, a, ename(a,e), *nested*).
abstractElement(a,e, ia)	coarsening(”, a, ename(a,e), *nested*)
addGate(a,e,g)	nestedExtension(ename(ae,e), *element*, gname(a,e,g), *gate*)
removeGate(a,e,g)	nestedCancellation(ename(a,e), *element*, gname(a,e,g), *gate*)
addLink(a,se,sg, te,tg)	refinement(”, gname(a,se,sg), gname(a,te,tg), *link*)
removeLink(a,se,sg, te,tg)	coarsening(”, gname(a,se,sg), gname(a,te,tg), *link*)
addBinding(sa,se,sg, ta,te,tg)	refinement(”, gname(sa,se,sg), gname(ta,te,tg), *binding*)
removeBinding(sa,se,sg, ta,te,tg)	coarsening(”, gname(sa,se,sg), gname(ta,te,tg), *binding*)

The syntax for the contract types is the one presented in section 2.3. The types are in italic font. We assume the existence of a function ename(a,e) which receives the name of an architecture and the name of the element and returns a name for the element that uniquely identifies it inside the model. This function can do something very simple as concatenating the name of the architecture and the name of the element, separated with a dot (ename(a,e) = a.e). In a similar way, function gname(a,e,g) returns the name of a gate that uniquely identifies the gate in the model.

As we mentioned in section 2.2.2.2, all these contract types have conditions for their application. The preconditions will be used by the conflict detection engine, in particular, they are the basis for defining and detecting the conflicts that appear as a consequence of parallel evolution, which is the main concern of this work, and which will be tacked in Chapter 5. Table 4 shows which are the preconditions we defined for each of the Adela increments.

The lists of edge type constraints and preconditions are neither absolute nor complete. The definition of these constraints and preconditions varies depending on the conceptual framework they support. Moreover, we did not include all possible rules in this table. For example, we could have added a constraint on nesting relationships that states that the nesting relationships cannot have cycles. We omitted this one because, for efficiency reasons that involve the implementation of the conflict detection algorithm, we prefer to define this check as an evolution conflict.

Table 4. Preconditions for the increments

Operations	**Preconditions**
AddArchitecture(A)	A is not an architecture in the model.
RemoveArchitecture(A)	A is an architecture in the model and it is not related to other elements by any kind of edge.
CopyArchitecture(A,NA)	A is an architecture in the model. NA is not an architecture in the model.
AddElement(A,E)	A is an architecture in the model. E is not an element inside architecture A.
RemoveElement(A,E)	A is an architecture, E is an element nested inside A, E is not connected by an edge to any other entity but A.
MoveElement(AE,E,AI)	AE is an architecture, E is an element nested inside AE, E is not linked to any other entity inside AE, AI is an architecture and does not have any element with name E.
ImplementElement(AE,E,AI)	AE is an architecture, E is an element nested inside EA, E is not described by any architecture, AI is an architecture and does not describe any element.
AbstractElement(A,E,IA)	A is an architecture, E is an element nested inside A, E has a description called IA and there are no bindings between gates inside IA and gates in E.
AddGate(A,E,G)	A is an architecture, E is an element nested inside A and G is not a gate inside E.
RemoveGate(A,E,G)	A is an architecture, E is an element nested inside A and G

	is a gate nested inside E. G is not related to any other element but E.
AddLink(A,SE,SG,TE,TG)	A is an architecture, SE and TE are elements nested inside it, SG is a gate in SE and TG is a gate nested inside TE. There is no other link from SG to TG.
RemoveLink(A,SE,SG, TE,TG)	A is an architecture, SE is an element nested inside it and SG is a gate in SE. TE is an element nested inside A and TG is a gate in SE. There is a link that relates SG and TG.
AddBinding(SA,SE,SG, TA,TE,TG)	SA is an architecture, SE is an element nested inside it and SG is a gate in SE. TA is the architecture nested inside SE, TE is an element nested inside TA and TG is a gate in TE. There is no other binding from SG to TG.
RemoveBinding(SA,SE,SG, TA,TE,TG)	SA is an architecture, AE is an element nested inside it and SG is a gate in SE. TA is the architecture that implements SE, TE is an element nested inside it and TG is a gate in SE. There is a binding that relates SG and TG.

4.5 Implementation

The implementation of the interpreter of Adela in PROLOG relies on the RC framework developed by Tom Mens. It consists of three logical parts: syntax, semantic, domain specific information. The syntax module provides predicates to check the syntax of an Adela description. The domain specific information module provides all the domain specific information needed by the reuse contracts framework. The semantics module defines predicates that fire the RC framework predicates in the application of the increments and predicates that interpret the resulting graph as an architecture. There is a main predicate which receives an Adela description, checks its syntax, and calls a predicate that returns it semantics.

In the following subsections we will briefly describe some of the predicates defined in the semantics and domain specific information modules.

4.5.1 Semantics predicates

The main predicate of this module is

semanticsArchModelDeclaration(+ArchModelDeclaration, -ArchSemantics, -ConflictsSemantics)[4]

ArchModelDeclaration is of the form ModelName(ArchModelDef, ArchNames), where ArchModelDef contains the list of increments and ArchNames is a list of the names of the architectures that appear in the model description.

[4] The predicates are defined in prolog notation. + before an argument means that the argument has to be bound‘ when the predicate is called, - before an argument means that that argument will be filled in during the execution of the predicate.

The semantics of a model is split in two parts, namely the architectural semantics (SemanticsArch) and the conflicts (ConflictsSemantics). ArchSemantics is false if a model with that name has already been declared previously. Otherwise, it is equal to an association of the model name with the architectural semantics. ArchSemantics and ConflictsSemantics are defined by predicate semanticsArchModel.

```
semanticsArchModel(+ModelName(+ArchModelDef,+ArchNames), -ArchSemantics,
-ConflictsSemantics).
```

The architectural semantics of a model is a set of associations of architecture names to the semantics of those architectures referenced in the list ArchNames. ConflictsSemantics holds the set of conflicts that appear during the application of the increments in ArchModelDef. The predicate semanticsArchModel is defined as follows:

```
semanticsArchModel(Name(ArchModelDef,ArchNames),Semantics,Conflicts) :-
  createGraph(Name),
  applyIncrements(Name, ArchModelDef,Conflicts),
  semanticsArchitectures(Name, ArchNames, Semantics).
```

Depending on whether each increment is basic or parallel, the predicate ApplyIncrements relies on predicates applyParallelIncrement and applyBasicIncrement in order to apply the increments to the graph. ApplyParallelIncrement uses the predicate merge of the RC framework (described in section 2.3) for the simultaneous application of the increments.

```
% applyParallelIncrement(+ArchModelName, +Increment, +User1, +User2,  -Conflicts)
applyParallelIncrement(ArchModelName, parallel(Incr1,Incr2), User1, User2, Conflicts)
:-
  merge(ArchModelName, Incr1, Incr2, User1, User2, Conflicts).
```

applyBasicIncrement calls the predicate evolve provided by the framework (described in section 2.3).

```
% applyBasicIncrement(+ArchModelName, +Incr, +User,-Conflicts) :-
applyBasicIncrement(ArchModelName, Incr, User, Conflicts) :-
  evolve(ArchModelName, Incr, User, Conflicts).
```

As a result, the graph is updated in memory with the modifications described in ArchModelDef and the conflicts are stored in Conflicts.

The predicate semanticsArchitectures is a complex predicate that reads the graph in memory and interprets it as an architectural model. This is done by calling the predicate architecture:

```
architecture(+ArchModelName,+ArchitectureName,-ArchitectureDescriptionSemantics)
```

which reconstructs an architecture from the plain graph, by checking if there exists a node that represents that architecture in the graph, and looking for all the elements and links that belong to that architecture. ArchitectureDescriptionSemantics represents the semantics of an architecture, which is built out of a set of elements and an architectural configuration.

```
architecture(ArchModelName,ArchitectureName, ArchitectureDescriptionSemantics) :-
        node(ArchModelName,ArchitectureName,architecture,_),
        buildElements(ArchModelName,ArchitectureName,ElementList),
        buildConfiguration(ArchModelName, ArchitectureName, ElementList,
                Configuration),
        ArchitectureDescriptionSemantics = ArchitectureName(ElementList,Configuration).
```

4.5.2 Domain specific information module

The domain specific information module contains all the information that the RC framework needs in order to apply Adela increments and detect the specific conflicts of the domain of software architectures. It defines predicates that will be called from the RC framework.

We added the types *architecture*, *element* and *gate* as subtypes of *node* in the type hierarchy. The types *binding* and *link* were added as subtypes of *edge*:

```
subtype(node,architecture).
subtype(node,element).
subtype(node,gate).
subtype(edge,binding).
subtype(edge,link).
```

The increments are translated by means of the translate_contract_type predicate, we transcript here only some examples:

```
translate_contract_type(addArchitecture(ArchName),
        production(extension,[ArchName, architecture])).

translate_contract_type(copyArchitecture(ArchName1, ArchName2),
  production(copySubgraph,[ArchName1, ArchName2])).

translate_contract_type(addElement(ArchName,ElemName),
production(nestedExtension,[ArchName, architecture, ElemName, element])):-
elementName(ArchName,ElemName,UniqueElementName).
```

The predicate elementName receives the name of an architecture and the name of an element and returns a name for the element that uniquely identifies it within the scope of a graph. This is necessary because the scoping conventions we defined for Adela in section 4.3.1 are not valid in the underlying graph. In the graph the labels of the nodes uniquely identify them, and the original name of the element can be equal to the name of another element in a different architecture. Similarly, there is a predicate called gateName(ArchName, ElemName, GateName, UniqueGateName) that does the same for the names of gates.

The predicate edgeTypeConstraint allows defining the restrictions on the edges, as discussed in section 4.4.4.2. For example, the restrictions for edges of type *nested* state that they can connect gates to elements, elements to architectures (meaning that the nested element belongs to the architecture) and architectures to elements (meaning that the nested architecture is the internal description of the element). Any *nested* edge that does not satisfy these restrictions will raise an error that will be interpreted as an applicability conflict. This is expressed in the following clause:

```
edgeTypeconstraint(edge(Graph,_,NestedEntity,NestingEntity,nested,_), Error) :-
  (nodetype(Graph,NestedEntity,gate), nodetype(Graph,NestingEntity,element)) -> true;
  (nodetype(Graph,NestedEntity,element),  nodetype(Graph,NestingEntity,architecture))
-> true;
  (nodetype(Graph,NestedEntity,architecture),  nodetype(Graph,NestingEntity,element))
-> true;
  otherwise -> Error=nestingConstraint([NestedEntity, NestingEntity]).
```

We will take *link* edges as another example. An edges of type *link* can only connect a gate in an element with another gate in another element if both elements belong to the same architecture.

```
edgeTypeconstraint(edge(Graph,_,Gate1,Gate2,link,_), Error) :-
```

```
(nodetype(Graph,Gate1,gate), nodetype(Graph,Gate2,gate),
 nestedIn(Graph,Gate1,NestedElement1),
 nestedIn(Graph,Gate2,NestedElement2),
 nestedIn(Graph,NestedElement1,Arch),
 nestedIn(Graph,NestedElement2,Arch)
) -> true;
otherwise -> Error=linkingConstraint([Element, Gate]).
```

Finally, the predicate domainSpecificPreconditions enables the definition of the necessary constraints on the application of the increments. We will transcript the preconditions of the increments implementElement and addLink as examples.

The preconditions implementElement express that the architecture must not be the internal description of any other element and the element must not already have an internal description.

```
domainSpecificPreconditions(implementElement(Arch,Elem, Desc),G) :-
  elementName(Arch,Elem,Element)
 presentNode(G,Element,element),
 presentNode(G,Desc,architecture),
 % the architecture does not describe another element
 not (presentEdge(G,_,Desc,OtherElem),
     presentNode(G,OtherElem,element)),
 % the element was not described before
 not (presentEdge(G,AnotherDesc,Element,nested),
     presentNode(G,AnotherDesc,architecture)),
 % there are no cycles in the nestings
  not nestedIn(G,Element,Desc).
```

The preconditions for the addition of a link check whether the types of the nodes are correct and state that the link can be added only if the two gates belong to elements that are nested in the same architecture.

```
domainSpecificPreconditions(addLink(Arch,Elem1,Gate1,Elem2,Gate2),G) :-
  gateName(Arch,Elem1,Gate1,GateName1),
  gateName(Arch,Elem2,Gate2,GateName2),
  elemName(Arch,Elem1,ElemName1),
  elemName(Arch,Elem2,ElemName2),
 presentNode(G, GateName1, gate),
 presentNode(G, GateName2, gate),
 presentEdge(G,_,GateName1,ElemName1),
 presentEdge(G,_,ElemName1,Arch),
 presentEdge(G,_,GateName2,ElemName2),
 presentEdge(G,_,ElemName2,Arch).
```

All the increments have domain specific preconditions.

The RC framework calls the predicates translate_contract_type, domainSpecificPreconditions and edgeTypeConstraint in order to perform the modifications in the graph specified in the increments. Those predicates that have to do with the definition of new conflicts are the subject of section 5.4 in chapter 5.

4.6 Summary

In this chapter we presented the architecture description language Adela. We introduced the conceptual framework it supports, its syntax, semantics and implementation.

Syntactically, an Adela description is a sequence of increments, each of them representing a modification to an initially empty architectural model. An Adela increment can be basic, or parallel. Each basic increment has preconditions that have to be satisfied by the model before the increment is applied. If the preconditions are not satisfied a conflict is raised. A parallel increment represents the simultaneous application of more than one modification to the architectural model.

The semantics of an Adela description is defined in terms of an architectural model and a set of conflicts. The architectural model is the result of the application of the increments to an empty architectural model, and the conflicts are the conflicts generated during the application of the increments to the model. In this chapter we did not go into detail in the definition of the conflicts, since they are defined in Chapter 5.

The implementation of Adela relies in the implementation of the RC formalism in order to perform the actual modifications and detect conflicts.

5 EVOLUTION OF SOFTWARE ARCHITECTURES

We are interested in detecting undesired interactions among simultaneous but independent modifications of an architectural model expressed in Adela. These interactions appear when different modifications are merged into one architecture by means of the parallel increment provided by Adela. This is especially important in a context where many people work together, and simultaneously take decisions that may affect the same artefact.

First, as a motivation for the problem of evolution of software architectures, we will explore some of the change requests that arose for the case of the SOUL system, and discuss their consequences on SOUL's architecture. Later in this chapter we will describe some conflicts that can appear as a consequence of interactions between independent evolutions, and how they can be defined and detected in an architectural definition of a system.

5.1 Evolution of architectural descriptions

Let's consider the architecture of SOUL, as we introduced it in the previous chapter. First we will consider new requirements causing this system to evolve, and their consequent changes to the architecture. Next we will analyse which are the consequences of merging the resulting evolutions into one architectural description.

The first evolution refers to an improvement of the communication between the repository and the query interpreter. Simultaneously, some other developers are working on another change request, which is the addition of a cache to improve the performance of the system.

Improve the communication with the repository

Since the communication between the Query Interpreter and the Repository is not very well structured in the original architecture, one group of developers has been assigned the task of adding one simple layer that handles all the communication. A new element called repository interface represents this layer. It hides the fact that the repository is a simple storage structure simulating a more intelligent repository, in which new rules can be asserted, existent rules can be retracted, and matching clauses can be found.

The resulting architecture is depicted in Figure 20.

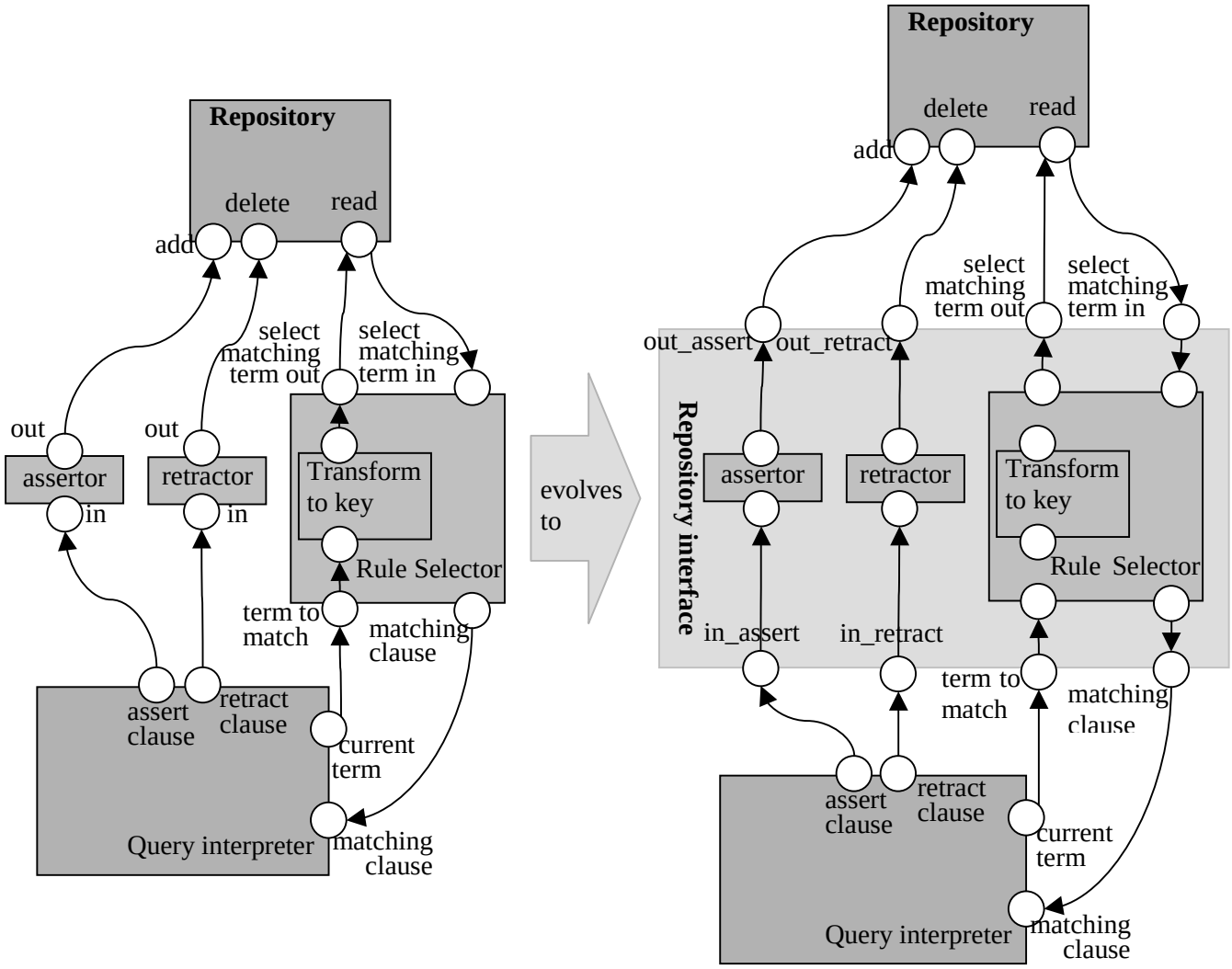

Figure 20. Adding a communication layer

The elements assertor, retractor, and rule selector now take part of the internal description of the repository interface element.

At the same time, another change for the same application is being designed.

Add a cache for rules and facts

As it is imperative to increase the performance of the interpreter, it is decided to cache the result of some rules. A cache is a storage structure that holds computed results for some of the rules. In order to be able to access those results the rule selector should communicate with the cache, and each time it has to find a match it looks in the cache first, before going to the original repository. Since the access to the cache is very fast and the results have been calculated already, if the needed clause is found in the cache the results are acquired very quickly.

Figure 21 shows a representation of the architecture after the needed modifications.

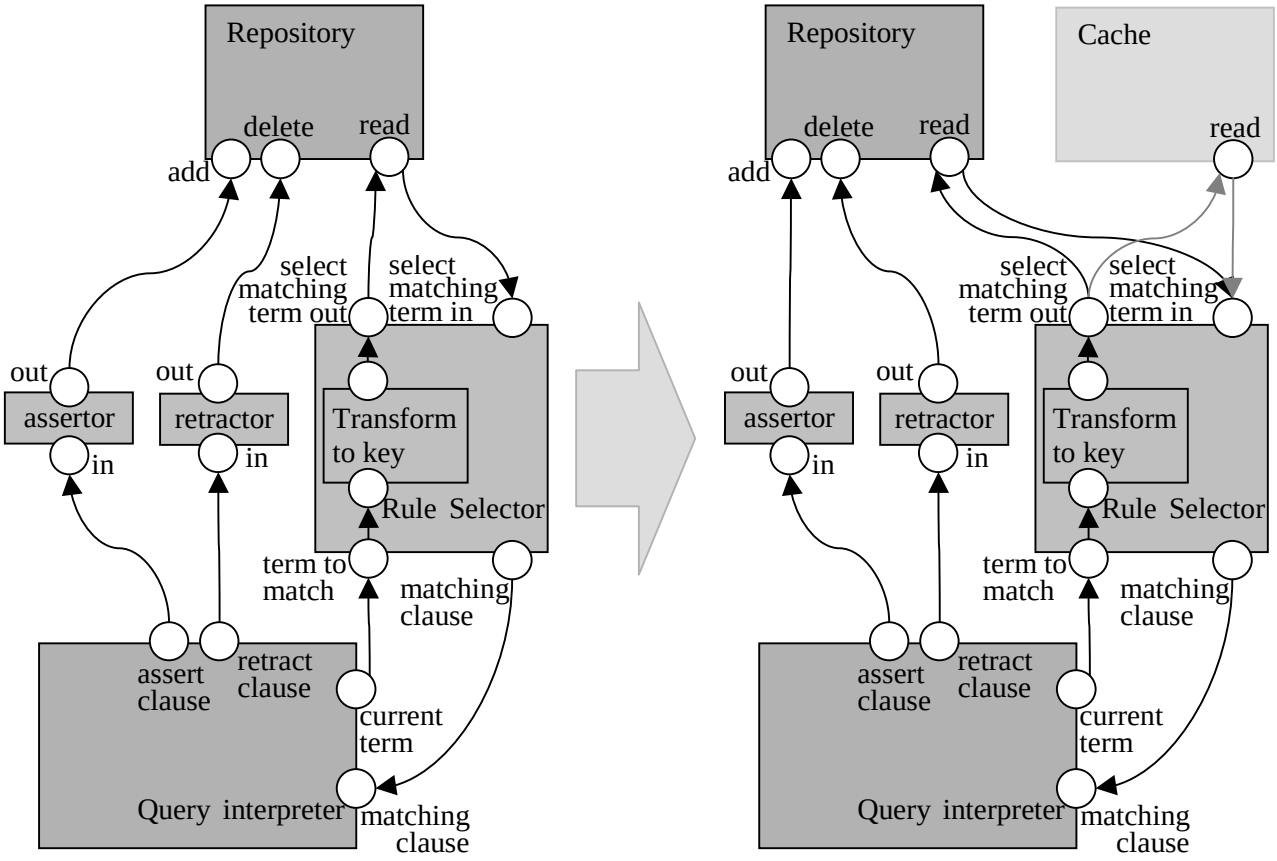

Figure 21. Adding a cache

This evolution is designed and performed in parallel with the previous one.

In the following section we will analyse what the consequences of the integration of both changes into one architectural description are.

5.1.1 Conflicts in evolution of architectures

In order to make any of these changes to the original architectural description, we need to express them as basic increments, write the whole evolution as an Adela program, and apply it to the architectural description.

Each of these two evolutions is legal: analysing them individually we can see they are both well designed so that the resulting architecture is one that can satisfy the requirements (e.g. the communication between the repository and the query interpreter is well defined in both cases). But, can they be safely merged together in the same architectural description? If we apply these evolutions one after the other, is what we obtain what we expect to obtain?

Looking a bit ahead, problems will arise with the communication between the query interpreter and the repositories of facts and rules (i.e. the original repository and the new cache), because following the ideas introduced by the first evolution (the improvement in the communication), this communication should be handled by the repository interface. However, since the cache was added and linked to the rule selector element, the

communication between the cache and the query interpreter will ignore the repository interface.

Given this scenario, we would say that there is a *conflict* between the two evolutions. The scenario is depicted in Figure 22. In this chapter we show how such conflicts can be detected based on an Adela description of the architecture and the evolutions.

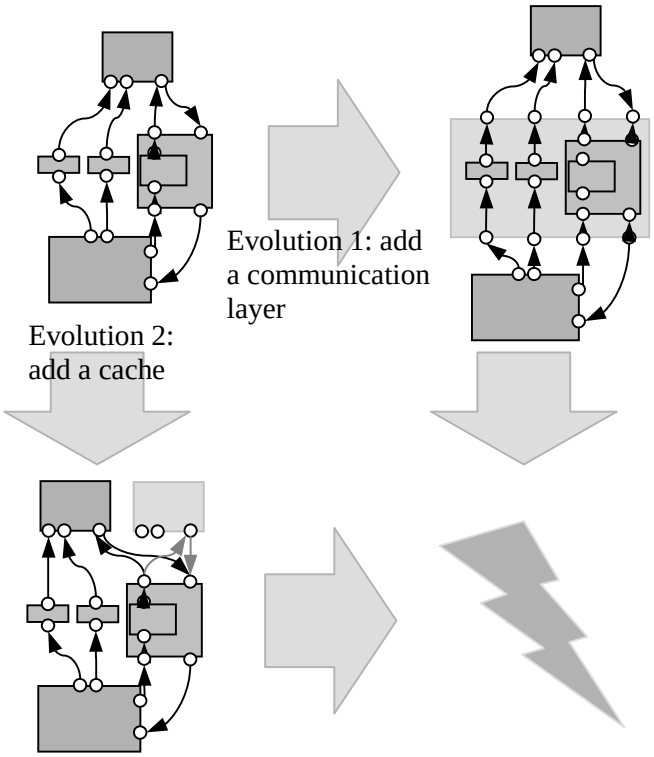

Figure 22. Conflicting evolutions

These kinds of mistakes and misunderstandings often occur in collaborative development environments. Being able to detect and warn about these situations avoids considerable loss of time and effort, and helps in assembling a coherent representation of the system. In some cases this lack of coordination leads to an invalid architecture, but sometimes only makes the architecture more confusing or redundant.

Note the similarities between this situation and the conflicts detected by reuse contracts. The same concepts can be applied. The software artefact would be, in this case, the architectural description. We already saw how the increments can be written in terms of contract types.

Recall that reuse contracts distinguish between different kinds of conflicts between two independent evolutions:

- Applicability (syntactic) conflicts are the conflicts that appear when one of the evolutions leaves the architectural description in a state that does not satisfy the preconditions of the other increment. We will analyse the increments and their preconditions in order to detect which increments conflict with the others.
- Evolution (semantic) conflicts are a more subtle kind of conflicts. Two evolutions, and even the merge of the two evolutions, may be legal in terms of the preconditions and give rise to a (syntactically) valid architecture. But there is a chance that the result is not the

intended one. The situations that are likely to represent a problem are captured by conflict patterns. Evolution conflicts are detected by looking for these 'dangerous patterns' in the graph. We have to identify which are the potential problematic situations in the case of software architectures.

The situation in the example will be detected as an applicability conflict. In order to perform the first evolution the repository interface element will be added (addElement(soul_architecture,repository_interface), addArchitecture(rep_architecture), implementElement(soul_architecture,repository_interface,rep_architecture)), and the rule selector element (together with the assertor and the retractor) will be moved inside the architecture that describes the element repository interface (moveElement(soul_architecture, rule_selector,rep_architecture)). The second evolution tries to add a gate to the rule selector, using the increment addLink(soul_architecture, cache, read, rule_selector, select_matching_term_in) and this conflicts with the moveElement increment (because if the element is moved, then it will not be found in order to add the link to it). This is a consequence of the interaction of the operations addLink and moveElement, and this conflict is called *Undefined Element in Link*, and it is defined in the next section.

5.2 Applicability Conflicts

Suppose the rule selector element does not have an internal description and two different users define different internal descriptions for it. Both increments are valid when considered in isolation, but when they are merged in the same architectural model they produce a conflict: the element has two different internal descriptions. This is another example of an applicability conflict.

In general, consider the architectural description of a system and two operations whose preconditions are satisfied in the architecture. Recall that the applicability conflicts appear when the application of one of the operations leaves the architecture in a state that does not satisfy the preconditions for the other. We will enumerate the conflicts we found for the operations we defined, and for each conflict we will explain which is the precondition that is broken. Table 5 relates the conflicts with the operations that provoke them.[5]

AC1: Duplicate Architecture

This conflict occurs when two different modifiers try to add an architecture with the same name to the model, i.e. the interaction of the operations AddArchitecture(Arch) and AddArchitecture(Arch).

One precondition for AddArchitecture(Arch) is that A is not an architecture in the model, and after performing one AddArchitecture(Arch) the precondition is not valid any longer for the other to be applied.

[5] Even if we believe the enumeration is exhaustive, we do not provide a formal prove of it. However a proof would be built in the same way Tom Mens proved that the conflicts he identified are the only ones that are possible. He did it by pairwise examination of all the contract types, and identifying those that are not parallelly independent, i.e. the precondition of the application of one production is preserved by the application of the other [Mens99].

AC2: Double Architecture Cancellation

This conflict appears when two modifiers try to remove the same architecture from the model. They both perform RemoveArchitecture(Arch).

A precondition for this operation is that Arch is an architecture in the model, and this is not true anymore after the first application of the operation.

AC3: Duplicate Element

Duplicate Element appears when there are two operations trying to add to the same architecture an element with the same name. The following pairs of evolutions can cause this conflict:

AddElement(Arch,Elem) – AddElement(Arch,Elem)
AddElement(Arch,Elem) – MoveElement(SourceArch,Elem,Arch)
MoveElement(SourceArch1,Elem,Arch)-MoveElement(SourceArch2,Elem,Arch)

It is a precondition of all the above operations that Elem is not an element in architecture Arch, and it is a consequence of all the operations that Elem is created as an element in the scope of arch. So no pair of these operations can be applied sequentially.

AC4: Double Element Cancellation

Double Element Cancellation is the conflict that is raised when the same element is being removed twice by two different modifiers from the same architecture.

It can be generated by the simultaneous application of:

RemoveElement(Arch, Elem) - RemoveElement(Arch,Elem)
MoveElement(Arch, Elem,TargetArch1)-MoveElement(Arch,Elem,TargetArch2)
MoveElement(Arch, Elem,TargetArch1) - RemoveElement(Arch,Elem)

A precondition for MoveElement(Arch, Elem,TargetArch1) and removeElement(Arch,Elem) is that Elem is an element in Arch. But once any of these this operation is applied, this condition is not satisfied any more, so the second application is not possible.

AC5: Undefined Element In Gate Extension

This conflict is raised when a modifier tries to remove an element and at the same time another modifier adds a gate to that element. This occurs when one of the operations removeElement(Arch,Elem) or MoveElement(Arch, Elem,TargetArch) and the operation addGate(Arch, Elem, Gate) are combined or merged.

A precondition for addGate(Arch, Elem, Gate) is that the element Elem is defined in the architecture Arch, and this precondition is never met after applying removeElement(Arch,Elem) or MoveElement(Arch, Elem,TargetArch1). On the other side, if addGate is applied first, it would create a gate nested inside Elem, breaking the precondition of removeElement(Arch,Elem) and MoveElement(Arch, Elem,TargetArch1) that states that Elem should not have any entity nested inside.

AC6: Undefined Architecture

This is the conflict that arises when a modifier tries to remove an architecture, and meanwhile another one tries to add an element to it. The pairs of operations that can generate this conflict are:

removeArchitecture(Arch) – addElement(Arch, Elem)
removeArchitecture(Arch) – moveElement(SourceArch, Elem, Arch)

If removeArchitecture(Arch) is applied first, then the precondition of addElement(Arch, Elem) and moveElement(SourceArch, Elem, Arch) that states that Arch has to be an architecture in the model is not met.

If addElement(Arch, Elem) or moveElement(SourceArch, Elem, Arch) is applied first, then removeArchitecture cannot be applied because the architecture to be removed has an element nested inside.

AC7: Undefined Architecture in Element Description

Arises when an architecture that is being added to an element as its internal description is at the same time being deleted by a different modifier. The operations that generate it are:

RemoveArchitecture(Arch) – RefineElement(HostArch, Elem, Arch)

The problem in this case is that once the architecture Arch has been removed, it is not possible to use it as the internal description of an element.

AC8: Undefined Element Description

Similar to the previous one, but in this case the element is deleted or moved by a modifier while it is implemented by another one. It is generated by any of the following combinations of operations:

RemoveElement(HostArch,Elem) – ImplementElement(HostArch, Elem, Arch)
MoveElement(HostArch,Elem,TargetArch) – ImplementElement(HostArch, Elem, Arch)

If RemoveElement(HostArch,Elem) or MoveElement(HostArch,Elem,TargetArch) is applied first then Elem is not an element of HostArch, and this is one precondition to ImplementElement(HostArch, Elem, Arch).

On the other hand, if ImplementElement(HostArch, Elem, Arch) is applied first, then a new edge is created starting from element Elem, so the preconditions of RemoveElement(HostArch,Elem) and MoveElement(HostArch,Elem,TargetArch) are not satisfied.

AC9/11: Undefined Link/Binding Source

Occurs when one modifier deletes a gate and another one adds a link with that gate as source, i.e. when RemoveGate(Arch,Elem,Gate) and AddLink(Arch, Elem, Gate, TargetElem, TargetGate) are performed in parallel.

Once RemoveGate is applied the gate is not present in the element any more, so it is not possible to add a link to it. Once a link is added to a gate, the preconditions for RemoveGate are no longer met.

Undefined Binding Source is very similar and appears when one modifier adds a binding, and another one deletes the gate that is the source of the binding. The operations that generate it

are RemoveGate(Arch, Elem, Gate) and AddBinding(Arch, Elem, Gate, TargetArch, TargetElem, TargetGate) when performed in parallel.

AC10/12: Undefined Link/Binding Target

Undefined Link Target is similar to Undefined Link Source but the deleted gate is the target of the link, i.e. this conflict occurs when RemoveGate(Arch,Elem,Gate) and AddLink(Arch, SourceElem, SourceGate, Elem Gate) are performed in parallel.

The precondition for AddLink(Arch, SourceElem, SourceGate, Elem Gate) which states that Gate has to be nested inside Elem is broken by RemoveGate(Arch,Elem,Gate), which deleted the gate. The precondition of RemoveGate(Arch,Elem,Gate) which states that the gate must not have connections is broken by the application of AddLink(Arch, SourceElem, SourceGate, Elem, Gate).

Undefined Binding Target is similar to Undefined Link Target, but the target gate of the binding is removed. The operations that generate it are RemoveGate(Arch,Elem,Gate) and AddBinding(SourceArch, SourceElem, SourceGate, Arch, Elem, Gate) when performed in parallel.

AC13: Double Element Description

Two modifiers that try to attach an internal description to an element generate this conflict. The interaction of the two operations ImplementElement(Arch, Elem,ImplementingArch1) and ImplementElement(Arch, Elem, ImplementingArch2) cause it.

One element can have only one description, so a precondition for attaching an implementation to an element is that there is no previous implementation for it. Of course, once one of these operations is performed, the preconditions for the other are not met.

AC14: Undefined Description

Undefined Description is the conflict that occurs when one modifier makes en element abstract and another one tries to add a new binding to it. The operations that cause it are AbstractElement(Arch, Elem, NestedArch) and addBinding(Arch, Elem, Gate, NestedArch, NestedElem, NestedGate).

If AbstractElement is applied first, then the effect is that Elem does not have an internal description any more, and so the preconditions for the addBinding are not satisfied. If addBinding is applied first, the new binding is created and so the preconditions for abstractElement are not satisfied.

AC15: Double Abstraction

This conflict appears when two modifiers try to make the same element abstract, i.e. when the two operations abstractElement(Arch, Elem, NestedArch) and abstractElement(Arch, Elem, NestedArch) are applied simultaneously. The precondition that is broken is the one that states that in order to perform abstractElement(Arch, Elem, NestedArch) the element Elem has to be implemented by the architecture NestedArch. When one of these operations is applied, this precondition is no longer satisfied for the application of the other.

AC16/17: Undefined Element In Binding/Link

Conflicts generated by the interaction of the operation MoveElement(Arch,Elem,TargetArch) with any of the operations AddBinding(Arch, Elem, Gate, NestedArch, NestedElem, NestedGate) or AddLink(Arch, Elem, Gate, TargetElem, TargetGate) because one modifier moves an element that another one is trying to address.

If MoveElement(Arch,Elem,TargetArch) is performed first then the element will not belong to the architecture arch any more, so AddBinding (Arch, Elem, Gate, NestedArch, NestedElem, NestedGate) or AddLink(Arch, Elem, Gate, TargetElem, TargetGate) cannot be applied. On the other hand, if AddBinding or AddLink is performed first then the element cannot be moved because it Elem is linked to other entities.

AC18: Undefined Element In Element Abstraction

Undefined Element in Element Abstraction is similar to Undefined Element in Element Description, but in this case the element is deleted or moved by a modifier while it is disconnected from its description by another one. It is generated by any of the following combination of operations:

RemoveElement(HostArch,Elem) – AbstractElement(HostArch, Elem)
MoveElement(HostArch, Elem, TargetArch) – AbstractElement(HostArch, Elem, Arch)

If RemoveElement(HostArch, Elem) or moveElement(HostArch, Elem, TargetArch) is applied first then Elem is not an element of HostArch, and this is one of the preconditions to AbstractElement(HostArch, Elem,Arch).

If AbstractElement(HostArch, Elem, Arch) is applied first then the conflict is not detected.

AC19/20: Double Link/Binding

Double Link is caused by the interaction of the operations

AddLink (Arch,SourceElem,SourceGate,TargetElem,TargetGate)
– AddLink (Arch,SourceElem,SourceGate,TargetElem,TargetGate)

because after one of them is applied, the precondition that states that there should not be any other link between SourceGate and TargetGate is no longer satisfied.

Double Binding is similar, but the operations that interact are

AddBinding (Arch, SourceElem, SourceGate, NestedArch, TargetElem, TargetGate) –
AddBinding (Arch, SourceElem, SourceGate, NestedArch, TargetElem, TargetGate)

Table 5. Table of conflicts

	AddArchitecture (a)	**RemoveArchitecture (a)**	**AddElement(a,e)**	**RemoveElement (a,e)**	**MoveElement (sa,e,ta)**	**ImplementElement (ae,e,a)**	**AbstractElement (ae,e)**	**AddGate (a,e,g)**	**RemoveGate(a,e,g)**	**AddLink (a,se,sg,te,tg)**	**RemoveLink (a,se,sg,te,tg)**	**AddBinding (sa,se,sg,ta,te,tg)**	**RemoveBinding(sa,se,sg,ta,te,tg)**
AddArchitecture (a)	AC1	-	-	-	-	-	-	-	-	-	-	-	-
RemoveArchitecture (a)	-	AC2	AC3	-	AC6	AC7	-	-	-	-	-	-	-
AddElement (a,e)	-	AC6	AC3	-	AC3	-	-	-	-	-	-	-	-
RemoveElement (a,e)	-	-	-	AC4	AC4	AC8	AC18	AC5	-	-	-	-	-
MoveElement (sa,e,ta)	-	AC6	AC3	AC4	AC3	AC8	AC18	AC5	-	AC17	-	AC16	-
ImplementElement (ae,e,ai)	-	AC7	-	AC8	AC8	AC13	-	-	-	-	-	-	-
AbstractElement (ae,e)	-	-	-	AC18	AC18 ae=sa	-	AC15	-	-	-	AC14	-	-
AddGate (a,e,g)	-	-	-	AC5	AC5	-	-	-	-	-	-	-	-
RemoveGate (a,e,g)	-	-	-	-	-	-	-	-	-	AC9 se=e AC10 te=e	-	AC11 se=e AC12 te=e	-
AddLink (sa,se,sg,se,te,tg)	-	-	-	-	AC17 se=e	-	-	-	AC9 se=e AC10 te=e	AC19	-	-	-
RemoveLink (sa,se,sg,se,te,tg)	-	-	-	-	-	-	-	-	-	-	-	-	-
AddBinding (sa,se,sg,ta,te,tg)	-	-	-	-	AC16	-	AC14	-	AC11 se=e AC12 te=e	-	-	AC20	-
RemoveBindin (sa,se,sg,ta,te,tg)	-	-	-	-	-	-	-	-	-	-	-	-	-
	AddArchitecture (a)	**RemoveArchitecture (a)**	**AddElement(a,e)**	**RemoveElement (a,e)**	**MoveElement (sa,e,ta)**	**ImplementElement (ae,e,a)**	**AbstractElement (ae,e)**	**AddGate (a,e,g)**	**RemoveGate(a,e,g)**	**AddLink (a,se,sg,te,tg)**	**RemoveLink (a,se,sg,te,tg)**	**AddBinding (sa,se,sg,ta,te,tg)**	**RemoveBinding(sa,se,sg,ta,te,tg)**

5.3 Evolution Conflicts

There is another kind of conflicts, which are different from applicability conflicts in that they do not break any precondition, but may be suspected of being an error. They are called evolution conflicts. When there is an evolution conflict, the operations requested can be performed, and the graph stays in a legal state, but here is a risk of an error. So these conflicts should be considered as warnings.

Evolution conflicts cannot be detected as violations of the preconditions, they can only be detected by looking at the resulting graph, at the different modifications that acted on it and the developers that requested those modifications.

We present here some examples of evolution conflicts that were discovered doing the experiments. This is not an exhaustive list, since the evolution conflicts are motivated by the practice rather than by the examination of the operations. New evolution conflicts can be found by experimenting with a different case, or considering different evolutions for the SOUL system.

Each of the evolution conflicts presented in this section is motivated by an example of a situation in which this conflict can appear.

EC1: Double Reachability

Assume that there are two groups of people working on the design of the cache for SOUL. The first group is in charge of the cache itself, while another group is working on the changes that have to be introduced in the rule selector element. Independently of each other both groups realise that there is the need for some communication between the cache and the rule selector, so they both add gates to the cache element and links in order to provide this communication.

The group in charge of the cache adds a single gate to it, handling both the incoming request and the returned result, and link it to the in and out gates of the rule selector. On the other hand, the group in charge of the rule selector element decides that two gates will be needed in the cache, one for the incoming request ad one for the returned result, and adds them together with the links to the rule selector element.

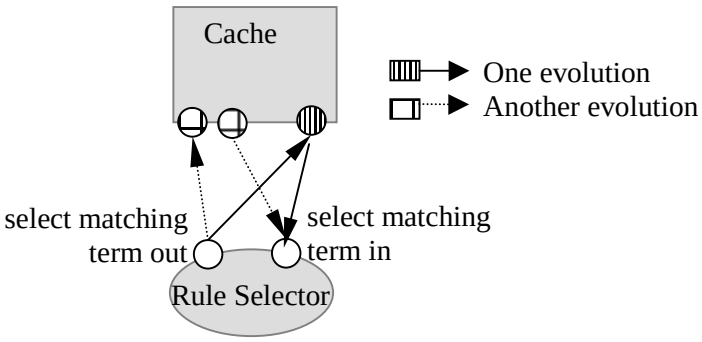

Figure 23. Example of double reachability

As a result we obtain an architecture in which there are redundant paths of communication between a pair of elements, as Figure 23 depicts. We would like to be able to detect this kind of situations, and warn the developers about this potential conflict.

It is important to notice that this situation is an error in this particular context, but might be correct in a different one, where both paths may be needed. However, it is valuable to detect it anyway and to show it as a warning.

Double Reachability is a conflict defined in Tom Mens' thesis. It occurs when two users add parallel edges, i.e. edges that relate the same pair of nodes [Mens99]. In our case, according to this definition, we would only get a double reachability conflict when there are two gates connected by two parallel links, or two parallel bindings. But unfortunately this would not allow us to detect really interesting situations. For example, with this approach we would not be able to detect the situation described above, since the links relate different gates. Figure 24 illustrates the original formulation of double reachability compared with the situation we would like to detect.

Figure 24. Double reachability: original formulation versus derived edges

In order to detect this more complex situation, we defined auxiliary derived edges between elements. There is a derived edge between element `a` and element `b` if `a` has a gate `a1`, and `b` has a gate `b1` and `a1` is related to `b1` by a link.

When the conflict detection algorithm takes into account these derived edges, conflicts like the one presented above in Figure 24 can be detected.

Furthermore, we could even want to detect situations such as the one in Figure 25, in which two users define different paths of communication among the same pair or elements, and each path can involve more than one link. In order to do this, we would need to extend the definition of derived edges in order to consider transitive edges as well.

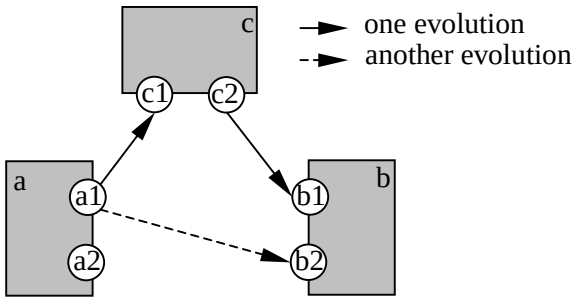

Figure 25. Double reachability with transitive edges

EC2: Inconsistent Moving

Recall the addition of a cache to the SOUL system. There are two possible solutions so that the rule selector can handle the cache. One of them is to change the internal description of the rule selector, so that it handles the cache. The other one is to create another element that has the rule selector as part of its internal description. These two possible solutions are shown in Figure 26. Now, what happens if both modifications are merged by mistake into the same architecture? We want to detect these situations in which a user modifies the description of an element (in this case the rule selector) and at the same time a different user moves the element to another architecture.

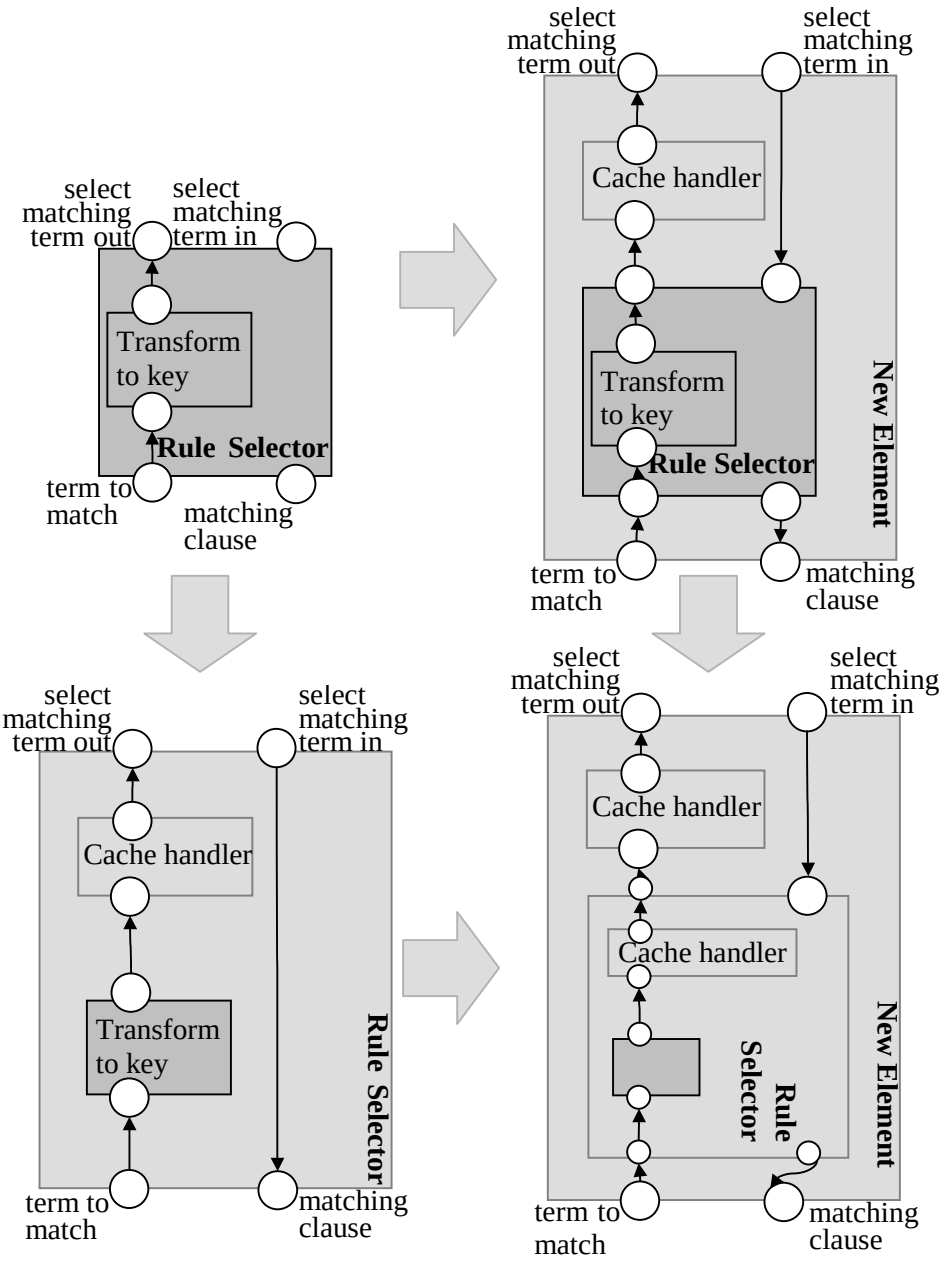

Figure 26. Inconsistent moving

We call this conflict Inconsistent Moving because the result is that the element is moved together with the architecture that describes it, but the description it carries with it is not the expected one. This situation is likely to be a mistake since the new description for the element may be useful only in the context of the original architecture. In this example, the new implementation of rule selector made no sense when we moved it to the new element.

This conflict occurs when one user performs MoveElement(OldHost, MovedElem, NewHost) and another one performs at least one of the following operations addElement(Arch, Elem), removeElement(Arch, Elem), addGate(Arch, Elem, Gate), removeGate(Arch, Elem, Gate), addLink(Arch, Elem, Gate, TElem, TGate), removeLink(Arch, Elem, Gate, TElem, TGate), moveElement(Arch, Elem, HostArch), if Arch is the name of the architecture that describes the element MovedElem.

It is detected by examining the resulting graph, looking at those elements whose description is changed by one developer and which were moved from an architecture to another by another developer. In order to see if the element was moved we need to look at nesting arrows that have been removed.

EC3: Inconsistent architecture modification

Consider an element E which is described by an architecture called A. Suppose that many changes have to be made to A. In order to perform this, one alternative is to change the current description of the element, adding new elements, deleting unnecessary ones, etc. The second alternative is to make the element abstract, by disconnecting its implementation, and make a new description for it from scratch in its place. Let's consider the scenario depicted in Figure 27: one developer thinks that the actual description will not be used, and disconnect it from the element, with the intention of making a new one for it. Meanwhile, another user designs and applies the changes that are needed to transform the old description into the new one. When these two evolutions interact, what we obtain is an element without description, and an architecture that would fit to the needed description, but it is not connected to any element.

Inconsistent architecture modification is a conflict that captures this situation. It occurs when a modifier makes a change to one element, gate or link inside the architecture that describes an element, and in the meantime another modifier makes the element abstract by disconnecting the internal description from the element. This conflict is likely to indicate a problem because the developer that designs changes for the internal description of an element may not know that the modified architecture will not describe that element any longer.

This conflict is caused by the interaction of AbstractElement(EArch, Elem) with one of the operations addElement(Arch, Elem), removeElement(Arch, Elem), addGate(Arch, Elem, Gate), removeGate(Arch, Elem, Gate), addLink(Arch, Elem, Gate, TElem, TGate), removeLink(Arch, Elem, Gate, TElem, TGate), moveElement(Arch, Elem, HostArch), if Arch is the name of the architecture that implements the element elem. It can be detected in the same way Inconsistent Moving Conflict can be detected.

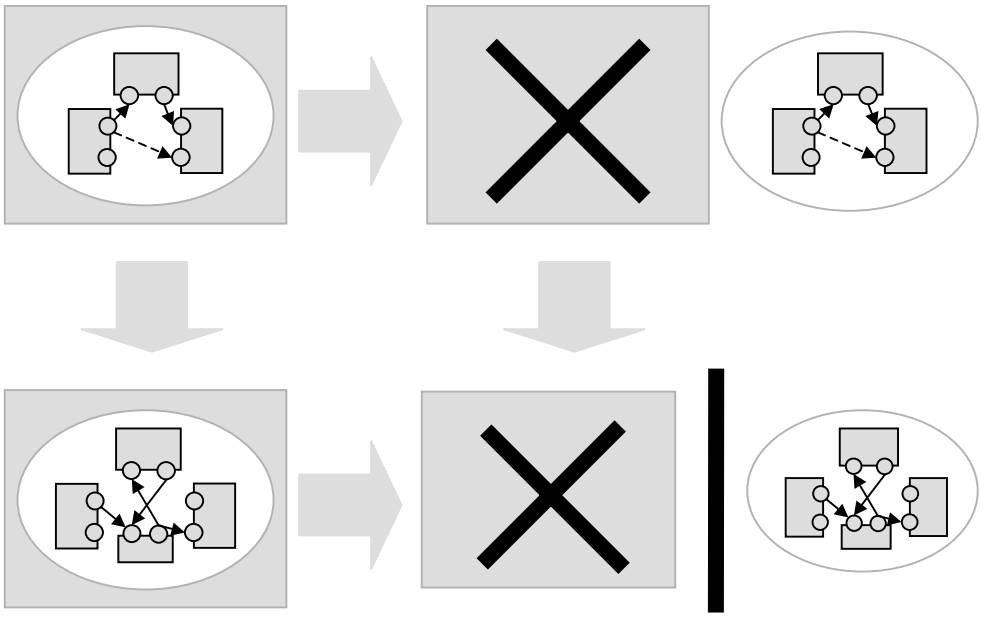

Figure 27. Inconsistent architecture modification

EC4: Cycle introduction

Sometimes the occurrence of cycles can indicate the presence of a problem in the design of an architecture. As an example, consider the architecture depicted in Figure 28. This architectural description expresses, as a consequence of the modifications proposed by two different developers, that an element called a creates an element called b, and element b creates element a. This situation is not feasible, and it is likely to be a mistake. Another example of cycles that represent a problem are the cycles of nesting edges.

Cycle introduction conflict detects these situations.

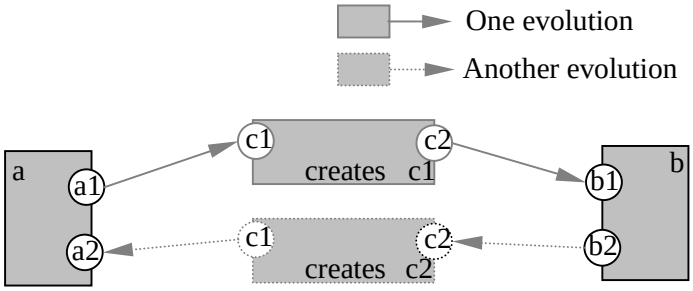

Figure 28. Cycle introduction conflict

Cycle introduction conflict had been defined in Tom Mens dissertation, but like in the case of reachability conflict the original definition of cycle introduction conflict only considered very elementary cycles, like the one in Figure 29.

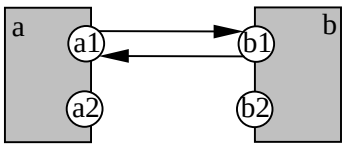

Figure 29. Original cycle introduction conflict

With the help of derived edges, conflicts like the one illustrated in Figure 28 can be detected.

5.4 Implementation

In this section we will describe how we customised the RC framework in order to detect the conflicts we described above. Since their implementation is very different, we separately analyse the implementation of applicability conflicts and evolution conflicts.

5.4.1 Applicability conflicts

The existent applicability conflicts can be renamed with more meaningful names, or names that are more appropriate to the specific domain of software architectures.

When a new operation is added, and once the interaction between the new operation and each of the existent ones have been analysed, it is very easy to extend the table with new applicability conflicts.[6] In order to illustrate the structure of the table of conflicts, let's consider the example of the definition of the conflict Double Element Cancellation conflict.

```
table(M,Tag,moveElement,moveElement,doubleElementCancellation).
```

5.4.2 Evolution conflicts

As an example of a conflict pattern, we can consider the definition of the Double Reachability conflict:

```
conflictPattern(G,doubleReachability(Tag,edge(G,E,U,V,T,C),Tag2,edge(G,E2,U,V,T2,
C2))) :-
  edge(G,E,U,V,T,C), edge(G,E2,U,V,T2,C2),
  typeRestriction(T,T2),
  differentModifier(C,C2,Tag,Tag2).
```

This predicate states that two edges follow the double reachability conflict pattern if they both connect node U with node V in graph G, they both have the same type (typeRestriction(T,T2)) and they were introduced by different developers (differentModifier(C,C2,Tag,Tag2)). In order to extend the pattern to incorporate derived edges, we changed the definition of the edge predicate, as follows:

```
edge(Graph,E,V,W,T,C) :- edge_fact(Graph,E,V,W,T,C).
edge(Graph,E,V,W,T,C) :- derived_edge(Graph,E,V,W,T,C).
```

[6] The table of conflicts was originally implemented in a different way, but we changed it in order to be able to add new entries in a modular way.

According to this definition, an edge can be a either a edge_fact or a derived_edge. edge_facts are facts that are asserted each time an edge is incorporated in the graph. There is a derived_edge between two elements if each of them has a gate and these gates are connected via an edge_fact:

```
derived_edge(G, '', SourceElement, TargetElement, derived, Const) :-
  nestedIn(G,SourceGate,SourceElement),
  nestedIn(G,TargetGate,TargetElement),
  presentEdgeFact(G,_,SourceGate,TargetGate,_,Const),
  presentNode(G, SourceElement, element),
  presentNode(G, TargetElement, element),
  presentNode(G, SourceGate, gate),
  presentNode(G, TargetGate, gate).
```

Some of the evolution conflicts defined in section 5.3 are implemented by defining the corresponding patterns.

5.5 Summary

In this chapter we analysed and presented the different kinds of conflicts we can detect. These conflicts are the consequence of unexpected interactions between independent modifications to an architectural model. They are classified in applicability conflicts and evolution conflicts. While applicability conflicts are found by comparing pairwise the preconditions of the increments, evolution conflicts are found by analysing different situations in the practice of architectural design and evolution that may lead to undesired results. We enumerated all the applicability conflicts that can occur due to the interaction of the increments in Adela, and presented some evolution conflicts we found in the experimentation with evolutions of the SOUL system. More evolution conflicts can be defined.

We finished the chapter by briefly describing the adaptations done to the RC framework in order to detect some of these conflicts.

6 EXTENSIONS TO THE LANGUAGE ADELA

The language Adela presented in Chapter 4 supports only very elementary operations. In order to prove that the approach is powerful enough to support more complex language constructs, in this chapter we will analyse some extensions to Adela. For each of the proposed extensions we will discuss which extra increments have to be added to the language in order to support this extension, how these increments can be expressed in terms of the basic contract types and what new conflicts we can detect based on the extra information provided. We will also make some comments on how these extensions can be implemented and integrated in the PROLOG prototype of the Adela interpreter.

While the first section is dedicated to the addition of several independent language features, the second one is concerned with a more fundamental need, i.e. support for reuse.

6.1 Different kinds of extensions

In this section we will consider several features Adela could extend with.

Recalling that the semantics of a description in Adela is defined in terms of an architectural model and a set of conflicts, the extensions can be classified in the following groups based on the impact they have on the syntax or semantics of the language:

- Syntactic extensions that make the *language more compact or readable* but do not affect the semantics of the language. In other words, these extensions are nothing more than *syntactic sugar* that make the language easier to use.
- Extensions that allow to detect *more and more specific conflicts* on the same conceptual framework for architectures, i.e. they enlarge only the set of conflicts that can be detected but do not affect the set of architectures that can be defined. This can be achieved by providing *more elaborated operations*, that capture more information on the intentions of the software architects. The conflict detection engine can use this information in order to detect more precise and more valuable conflicts. Note that these elaborated operations are *more* than syntactic sugar: although they also might make the language more compact, readable or easier to use, in addition they provide extra information that may be used to detect extra conflicts.
- Extensions that *enrich the conceptual framework for architectures* with more specific information that currently is not part of the model. For example, in order to address a more specific concern, such as the static decomposition of the system in modules, special kinds of nodes as well as checks on their interconnections may be needed. Of course, these extensions of the architectural model will induce the definition of new conflicts.

It is not the aim of this chapter to provide a complete enumeration and analysis of different extensions to Adela. Rather, we want to show how it is possible to extend Adela,

experimenting with some of the language features that appear more frequently in the literature and some others that seem to be very promising in terms of expressiveness.

6.1.1 Syntactical extensions

Several increments can be defined that make the language more compact without adding information to the model it supports or allowing the detection of new conflicts. Here we will briefly present some of them.

6.1.1.1 Direction of bindings

Recall that the increment addBinding(A,SE,SG,TA,TE,TG) is such that in order to express a bi-directional binding, two bindings in opposite directions had to be added. Moreover, in the signature of this operation it is not clear which is the internal gate, and which is the external one. To make the language clearer and the conflict detection easier, it is a good idea to make explicit the direction of the bindings. The direction will be expressed when the binding is added, and will be one of *in*, *out* or *inOut*. The binding is *in* if the information flows from the external gate to the internal one, it is *out* if the information flows from the internal gate to the external gate and it is *inOut* if the flow is bi-directional. This is merely syntactic sugar, because in fact the same modifications can be expressed by means of the original increments, and once a binding has been added the information about its direction can be extracted from the previous definition of the model by considering both the links and the nesting relationships. The new increments to add and remove bindings will be addBindingD(A, SE, SG, TA, TE, TG, Direction) and removeBindingD(A, E, G, NestedA, NestedE, NestedG), where Direction is one of in, out or inOut. The semantics of these operations is now more explicit and intuitive than the semantics of the original ones.

Notice that we could even define new increments based on this ones, such as changeBindingDirection(A,SE,SG,TA,TE,TG,Direction). This would allow us to detect more conflicts, like the one that would appear when two evolvers independently change the direction of a binding in an inconsistent way. When we do this we are actually going beyond syntactic sugar, we are extending the syntax of the language to allow more expressiveness, as well as to be able to check more conflicts.

Implementation

addBindingD(A,SE,SG,TA,TE,TG,in) could be implemented as addBinding(A, SE, SG, TA, TE, TG), addBindingD(A,SE,SG,TA,TE,TG,out) as addBinding(TA, TE, TG, SA, SE, SG) and addBindingD(A,SE,SG,TA,TE,TG,inOut) and be implemented as the sequence addBinding(A, SE, SG, TA, TE, TG), addBinding(TA, TE, TG, SA, SE, SG). However, this new syntax motivates a different mapping to the graph: we can have different subtypes of type *binding*, namely *in*, *out* and *inOut*, so that information is made explicit also in the graph. This will make the conflict detection algorithm easier. In this case, addBindingD(A, SE, SG, TA, TE, TG, *in*) is to be translated into the contract type refinement(", gname(A,SE,SG)[7], gname(A,TE,TG), *in*). addBindingD(A, SE, SG, TA, TE, TG, out) and addBindingD(A, SE, SG, TA, TE, TG, *inOut*) are to be translated to contract types in a similar way.

[7] Recall that gname(A,E,G) is a function that returns a name for the gate G that uniquely identifies it in the model.

The domainSpecificPreconditions of the addBinding increment check if the direction indicated is correct, if the gates and elements exist and if one of the elements is nested inside the architectural description of the other:

```
domainSpecificPreconditions( addBinding( NestingArch, NestingElement, NestingGate,
NestedArch, NestedElem, NestedGate, Direction),G) :-
% the direction in correct
  (Direction = in; Direction = out; Direction = inOut),
% the unique names of the elements and gates are calculated
  elementName(NestingArch, NestingElem, NestingElemName),
  elementName(NestedArch, NestedElem, NestedElemName),
  gateName(NestingArch, NestingElem,NestingGate, NestingGateName),
  gateName(NestedArch, NestedElem, NestedGate, NestedGateName),
% the types of the nodes are correct
  presentNode(G, NestedGateName, gate),
  presentEdge(G,_,NestedGateName,NestedElemName,nested),
  presentNode(G, NestingGateName, gate),
  presentEdge(G,_,NestingGateName,NestingElemName,nested),
  % one element is nested inside the other
  presentEdge(G,_,NestingElemName,NestedArch,nested),
  presentEdge(G,_,NestedArch,NestedElemName,nested).
```

6.1.1.2 Flow of links

An extension similar to the one for the direction of the bindings could be made for links. We could distinguish between uni-directional and bi-directional links. The increments to add a new Link would be addLink(A,SourceE,SourceG,TargetE,TargetG,Flow) where Flow is one of *uni* and *bi* . Again, this can be implemented by having different subtypes of the type *link*: *uni* and *bi* in a way similar to the implementation of direction in the bindings.

The increment changeLinkFlow(A, SourceE, SourceG, TargetE, TargetG, Flow) to change the flow of a link from *uni* to *bi* or from *bi* to *uni* could be added as well.

6.1.1.3 Adding multiple gates simultaneously

Some other increments can be thought of that make the language more compact. Instead of adding all gates to the element one by one, as was the case in the example presented in section 4.3.2, we could define an operation addGates that adds a set of gates to an element. The signature would be addGates(Arch, Elem, SetOfGates) and it could be implemented in terms of the repeated application of addGate(Arch, Elem, Gate). Notice that the translation of addGates(Arch,Elem,Set) would be of the form: for all the gates in the set perform addGate(Arch,Elem,Gate), and that the expression for all the elements in the set is not expressible in terms of contract types. At this point, we classified this increment as pure syntactic sugar, but maybe it could even allow detecting some more subtle evolution conflicts, based on the fact that the gates in the set of gates are added simultaneously as a whole (this information is now made explicit and was not available before)

6.1.1.4 Deletion of entire gates and elements

Another increment that would make descriptions more compact is one that removes a gate together with all the links and bindings that relate it with other gates. This increment called removeEntireGate (Arch,Elem,Gate) can be implemented as: for all the links where the gate is source or target perform removeLink, for all the bindings in which the gate is source or target

perform removeBinding, removeGate. It can generate any of the conflicts that any of these three increments can generate. Notice that we cannot know how many links or bindings will be deleted until we know which is the particular gate to delete, so the sequence of primitive contract types that will be needed to perform this increment is not fixed. Again, this means that there could be extra semantics in this increment, even though we did not find any specific conflict.

removeEntireGate can be used to implement a higher level increment removeEntireElement(Arch, Elem) that erases an element together with everything that is nested in it, and the links that relate it with any other element. This increment can be implemented by means of repeated application of removeEntireGate and removeElement. The set of conflicts this operation can raise is the union of the set of conflicts that the increments that implement it can raise. Again, for now we classified these increments that are mere groupings of other increments as syntactic sugar, but with more careful analysis, we might be able to detect more interesting, or higher level, conflicts based on the fact that the different constituting increments belong together.

6.1.1.5 Others syntactic extensions

Similar to removeEntireElement an operation called addEntireElement(Arch, Elem, SetOfGateNames) could be defined, that creates a new element with all its gates.

Another increment that is related to addEntireElement one is connectElement(Arch,Elem, SetOfPairsOfGates), that connects an element with others. This increment can be implemented by repeated application of addLink.

A very simple increment that, however, is very useful is renameElement. Renaming is discussed in Mens' dissertation [Mens99].

6.1.1.6 Remarks

Note that it is not always easy to decide whether some increment is merely syntactic sugar making the language easier to use, or whether it actually provides a more intentional description, thus allowing a conflict detection engine to detect more specific operations. As an example consider an operation such as rename which, at first sight, seems to be nothing more than syntactic sugar for a combination of 2 operations: addition and removal. However, by specifying them as one single operation that should be considered as a whole, we can better express the intention of the evolution, thus allowing us to detect more specific conflicts. Having made this observation, it might be the case that some increments that are classified here as mere syntactic sugar, could, with a bit more careful analysis, allow us to check more detailed conflicts, and thus should be categorised as extensions that allow to detect *more and more specific conflicts* on the same architectural model. However, as we said, the intention of this chapter is mainly to get a feeling of the kinds of possible extensions that can be made to Adela.

We can separate the increments we presented in this section in two groups. Some of them, like the ones that make explicit the direction in bindings or the flow in links, can be expressed very easily in terms of a couple of basic contract types. The others, removeEntireGate, removeEntireElement, addGates, connectElement, renameElement, are in fact groupings of an arbitrary number of operations. We believe that this second group is more likely to give rise to new conflicts.

6.1.2 Defining and detecting more conflicts

In addition to the examples of changeBindingDirection and changeLinkFlow, which were already mentioned in the previous subsection, in this subsection we will analyse the implementation of some other increments that seem to be useful in the modelling of software architectures and that enable the detection of new conflicts. In particular we will discuss increments to copy an entire architecture and to merge several elements into one.

6.1.2.1 Copying architectures

Often the same architectural pattern is repeated in several parts of a system. Since the language Adela does not provide any built in mechanism to reuse architectural descriptions, a first step in order to be able to reuse architectural descriptions is the implementation of an increment that produced a (deep) copy of an entire architecture to another part of the system. It is important to point out that sharing an architectural description without (deep) copying is not trivial. The reason is that the gates of the original architecture would be connected through bindings to a set of gates different from the ones of the reuser architecture, but there is only one set of gates. Thus it is not possible to differentiate between those links that belong to the original architecture from those ones that belong to the reuser. We propose the implementation of an operation copyArchitecture(OldArchitecture, newArchitecture) that creates an architecture similar to OldArchitecture, but under the new name newArchitecture. Note that this can only be a first step towards achieving reuse of architectural descriptions, because 'reuse by copy' can hardly be considered as 'good' reuse. For example, we loose the information of where did a copy came from after it has been copied, and changes to one of the copies are not checked against the original or other copies. In a later section (6.2) we will go into more depth on the problems of achieving reuse of architectural descriptions, and mention a more elaborated solution.

The operation copyArchitecture(OldArchitecture, newArchitecture) gives rise to evolution conflicts. For example when one modifier copies the architecture while another one makes changes inside it, a conflict should be triggered, since the effect of the interaction of these two evolutions is that the architecture copied is not the expected one.

Implementation

The effect of the increment copyArchitecture(Old, new) is the creation of a new architecture called new, whose elements and configuration are the same as those in Old, but which has a different name new. Not only the element nodes, but also their nested gates, have to be copied. If some of the elements have an internal description, then the entire architecture that represents the internal description has to be copied too. Thus, the translation of the operation copyArchitecture in terms of the other contract types is not easy. Instead, we preferred to add a new contract type to the framework, called copySubgraph(V,W). The only precondition for the application of this operation is that the node V must exist and the node W must not exist:

```
preconditions(production(copySubgraph,[V,W]),Graph) :-
  presentNode(Graph,V), absentNode(Graph,W).
```

The increment copyArchitecture is then defined in terms of copySubgraph:

```
translate_contract_type(copyArchitecture(ArchName1, ArchName2),
  production(copySubgraph,[ArchName1, ArchName2])).
```

And the only domain specific precondition for this increment is that the first argument must be an architecture:

```
domainSpecificPreconditions(copyArchitecture(Arch1,Arch2),G) :-
  presentNode(G, Arch1, architecture).
```

Besides the evolution conflict described above, copyArchitecture enables the detection of new applicability conflicts, since some of the existent increments conflict with this new increment. For example, addArchitecture(A,N) conflicts with copyArchitecture(A,N) because they both add an architecture with the same name. And removeArchitecture(A) conflicts with copyArchitecture(A,N) because the precondition that states that A must exist is not satisfied after the application of removeArchitecture(A).

Notice that the names of the elements in the original architecture are the same as the ones in the copy, only the name of the architecture is different. This does not contradict Adela naming conventions, but it does generate problems at the level of the graph, since the graph is one unique scope. This is solved changing the names of the elements when they are copied, in a way that is transparent to an Adela user.

6.1.2.2 Merging elements

A task that seems to occur frequently in the process of modelling the architecture of a system is merging two elements into one. The effect is that one of the original elements is deleted from the model and the other element's interface is enhanced with the gates of the first one. The gates carry the links with them. There are two restrictions: only the host element is allowed to have an internal description, and both elements have to be defined in the same architecture, so that the links are valid when the gates and links are moved from one element to another.

MergeElement (Arch, Elem1, Elem2, NewElem) can be implemented by means of the following increments: removeEntireElement, addEntireElement, connectElement. An evolution conflict is generated when a modifier merges two elements and another one adds a gate to the element that acts as a host in the merge. This is a potential problem because the user who adds a gate is not aware of the new gates that are being added to the element, namely the gates of the other element being merged, and maybe the gate he is adding is not needed in the new context.

6.1.3 Enriching the architectural model

As we said before, Adela is a very basic language in the sense that the architectural model it supports only has very basic features. Its architectural model can be enriched by

- distinguishing among different kinds of nodes, and by
- defining more specific interconnections between the nodes, i.e. different kinds of edges.

6.1.3.1 Nodes: components and connectors

Currently, Adela only supports one kind of elements, which are grouped in architectures and whose interfaces are described in terms of gates. However, recalling from section 3.1.2, in literature some authors recognise the need for distinguishing between *components* and *connectors*, components being the loci of computation and state and connectors being the

building blocks that govern interaction among components and rules that govern them [Medvidovic&Taylor97].

In order to distinguish between components and connectors some other changes may be needed. In the model proposed by Shaw and Garlan the gates in the components are called ports and the gates in the connectors are called roles. The components can only be directly connected to connectors, connectors can only be directly connected to components, and this is expressed by restrictions on the relationships between ports and roles: a role can be linked only to ports, a port can be linked only to roles. Moreover, roles can be related via bindings only to roles, and ports to ports [Garlan&Shaw96]. These restrictions impose a very disciplined use of these elements.

Recall that the semantics of an element is determined by its name, its set of gates, its internal description and a set of bindings that determine the relationships between the internal description and the external gates. Incorporating connectors and components in our original architectural model would affect the semantics of the elements because elements would now have an annotation that tells whether they are connectors or components. Also the semantics of gates would be enriched with the knowledge of being ports or roles.

The syntactic level of the language would also be affected. In order to differentiate between connectors and components, and between ports and roles we would need to add operations to add each of these elements and delete them: addComponent(Arch, Comp), addConnector(Arch, Conn), addPort(Arch, Comp,Port), addRole(Arch, Conn,Role). Some other operations may be needed, such as turning a component into a connector (becomeConnector(Arch, Comp)), which would involve changing the ports in that components into roles, or turning a connector into a component, which would involve turning the roles into ports.

Of course, as a consequence of having extended the syntax, and added the necessary increments working on the syntax, new conflicts arise. An example of an applicability conflict is the one that occurs when one user adds a port to a component and another one independently turns that component into a connector. This generates an applicability conflict because the consequence of merging these increments will be that the port will be added to a connector, violating the restriction that says that connectors communicate through roles.

Implementation

This feature can be implemented by explicitly adding the following new types: *component*, *connector*, *port* and *role*. In this case the translation of

addComponent(arch, comp)

is

addElement(Arch, Comp)
retypeNode(ename(Arch, Comp), *component*)[8]

or directly

nestedExtension(Arch, *architecture*, ename(Arch, Comp), *component*)

And the implementation of

[8] Recall that ename(A,E) is a function that returns a name for the element E that uniquely identifies it in the model.

becomeConnector(Arch, Comp)

is just

retypeNode(ename(Arch, Comp), *connector*)

The *link* edges would have special restrictions, such as that a *link* can relate a *port* with a *port* or a *role* with a *role*, but never a *port* with a *role*. The bindings will have as an additional restriction that no *binding* can relate a *role* with a *port*, or a *port* with a *role*.

The organisation of the type hierarchy can become rather complex when it has to represent orthogonal classifications. So it might be necessary to use other means of classification. Constraints can be used for this purpose. In this case, a node can have a constraint that states whether it is a component or a connector, and a gate can have a constraint that states whether it a port or a role.

So the translation of

addComponent(Arch, Comp)

is

addElement(Arch, Comp)
putConstraint(ename(Arch, Comp), *component*)

Since there was no support for the addition of constraints in the framework we added two contract types to the RC framework: one to add and one to delete a constraint from the list of constraints of a node.

6.1.3.2 Nodes: cardinalities

The gates in the elements can have cardinality constraints. For example, we might want to ensure that a certain gate should always be linked to another gate, or that a gate supports at most two links. This would allow us to express, for example, that a server has a certain limited capacity, i.e. it cannot answer requests from more than a certain number of clients.

In order to provide this functionality we can define the increments addGate(Arch, Elem, Gate, Cardinality). Cardinality can be one of atLeastOne, atMostOne, atLeastTwo, …

Cardinalities can be attached to gates by means of constraints.

6.1.3.3 Nodes: Types of elements

In the same way connectors and components can be defined, we can distinguish among other kinds of nodes. Each of these kinds of nodes can have certain predefined constraints, which are expressed as preconditions on the operations that introduce the new nodes in the model. This could be an interesting way of differentiating among pipes, filters, clients, servers, data repositories, etc. For example, we could define a pipe as a connectors that only can communicate with filter components, and which must have at least an incoming link in one of its gates and an outgoing link in one of its gates. We could also define a pipeline pipe as a special kind of pipe that has exactly two gates, one with one incoming link and one with an outgoing link.

Pipe, *filter*, *pipelinePipe*, *pipelineFilter*, *client*, *server*, *serviceRequest*, *dataRepository* could be represented as subtypes of *connector* or *component*. Their restrictions could be implemented as preconditions of the operations that deal with them. The increments that

introduce and eliminate elements have to be changed so that the kind of the element can be supplied. Also an operation changeType(Arch, Elem, OldType, NewType) can be supplied.

The conflicts we would be able to detect depend on these more specific types. Consider the following situation: in a pipe-and-filter architecture, a user considers that some of the pipes which only have one incoming and one outgoing link to a filter can be treated as pipeline pipes and changes their type to *pipelinePipe*. Another evolver, not aware of this decision, adds an extra outgoing link to a role in one of the pipes, so it does not comply to the pipeline restrictions any longer. This situation will produce a conflict.

6.1.3.4 Nodes: Styles

Many authors consider an architectural style as a language of different connectors and components together with some restrictions concerning their interconnection [Garlan&Shaw96]. In the way we have just explained it is possible to distinguish among several kinds of connectors and components, but we are not yet able to restrict an architecture to a certain style, i.e. to a certain subset of those kinds of connectors and components. We would like to be able to express in Adela that a certain architecture is a client-server architecture, meaning that its components are clients and servers communicated via service-request connectors. In order to do so, we have to distinguish among several kinds of architectures. This can be achieved by defining different subtypes of the *architecture* type. *clientServer*, *pipeline*, *pipes&filters* can be defined as types, and attached to the architecture nodes. The increment to introduce an architecture would now require the information about the style to which it belongs: addArchitecture(Arch, Style). The preconditions of the increments to introduce new elements in an architecture should ensure that only elements of the types allowed for the style of that architecture are introduced. For example, if architecture Arch is of style *clientServer*, meaning that only elements of type *client*, *server* and *serviceRequest* are permitted, the preconditions of addElement(Arch, Elem, KindOfElement) should require that KindOfElem is one of *client*, *server* and *serviceRequest*.

This approach, however, does not yet solve the problem of style in a completely satisfactory way. One important drawback is that each kind of element or style has to be added as a new type, and the preconditions and constraints have to be hard-coded. The implementation of the conflict checking tool has to be modified each time a new style is incorporated, making the addition of new kinds of elements and styles by a user difficult or impossible.

Moreover, this mechanism to represent styles seems to be quite restrictive. It is not possible to express more precise architectural configurations as styles. Consider the following example: we may want to define the basic architecture of a compiler, like the one presented in 3.1 as a style, i.e. a very general configuration that several specific compiler architectures will refine and complete. Such a style would have as components a parser, a scanner, a semantic analyser and a coder, as well as some predefined interrelationships among them. We would like to use this description as a starting point to define compilers. Moreover, given an evolution on one of the architectures derived from this basic configuration, we want to know whether it violates the basic configuration of the style or not. This is not solvable with the approach we are considering, because it is not possible to specify this kind of configurations. In section 6.2 another more radical solution for this problem is presented.

6.1.3.5 Edges: forbidden edges

Also the edges can be enhanced with more information. One very useful example is the definition of forbidden edges. Forbidden edges express that it is not allowed to have a link or

binding between the elements related by the negative edge. We could provide this functionality by adding the four increments addNegativeLink, removeNegativeLink, addNegativeBinding, removeNegativeBinding.

A simple example of a conflict that we would be able to detect with this information is when one user adds a negative link and another user adds a (normal) link between the same pair of nodes. This would cause an applicability conflict.

Negative edges can be implemented by adding in the constraints information that expresses whether an edge is negative or not, or by means of types, defining *forbidden* as a subtype of *edge*.

6.2 Reuse of architectural descriptions

The original formulation of Adela provides no support for reuse. It supports the evolution of a single architectural model into a new version, but it is not possible to come back to a previous state i.e. when a model evolves, the previous version is lost. In order to support reuse, and to be able to deal with families of related software assets it is necessary to be able to access to both the original asset and the reused version, and it should be possible to adapt a single reusable asset to different specific situations. In this section we will explore mechanisms that will enable the definition of an architecture starting from a previous description, keeping both the new and the original version. We will also explore some new possibilities that this new feature enables.

6.2.1 Modelling Reuse

Recalling section 2.2.2.5, reuse can be modelled in reuse contracts by explicitly documenting the modifications by means of edges of type *reuse* in the graph. When we reuse an asset we want to keep both the original and the new version, and we want the new version to comply with the original one. An evolution, in contrast, can make any modification on the system.

Given that we can document the reuse relationships of different architectural descriptions, we can think that these reuse edges together with the architectural models they relate constitute a hierarchy of architectural descriptions. The graph itself would act as a repository for architectural descriptions.

The preconditions and constraints we define for the increments that handle the reuse edge determine the semantics of the reuse relationship. We could state, for example, that in order for an architectural description to reuse another one, it must have at least all the nodes and edges from the first one, i.e. no remove operations are allowed. In this case, we would have a hierarchy of incremental architectural definitions. The new version of an architecture can add elements, gates, links, bindings, and even internal descriptions for elements, on the original architectural description. This would also give us a notion of refinement.

In order to provide support for reuse, we would need to define a new increment (mapped onto a new contract type):

reuse(OldModelName, NewModelName, increment)

where increment would be one of the increments we defined before, and would express the modification introduced by the new architectural description into the original one.

Another different increment would allow to evolve an architectural description:

evolve(ModelName, increment)

6.2.2 Reuse versus evolution conflicts

Given a hierarchy of architectural descriptions, and provided that the root architectural model evolves, we could answer whether the reuser descriptions still complies to the new version. All the conflicts we analysed so far are applicable in this situation. If an applicability conflict appears between the reuse relationship and the evolve relationship, it means that the reuser does not anymore comply with the provider. If an evolution conflict appears, it means that there is a potential problem that should be analysed. As a trivial example, consider that the model M is reused by model MReuser, which adds a link to it:

reuse(M, MReuser, addLink(A,E, G, TE, TG))

Some time later, the following evolution is fired for model M:

evolve(M, addElem(A,E,anotherG,TE,TG))

First the reuser added a link between elements E and TE via gates G and TG. Later, an evolution added a link between elements E and TE via gates anotherG and TG. In the reuser model, this is likely to be a redundant path of communication. A Double Reachability evolution conflict will indicate this problem.

The possibility of doing these checks is important in environments based on product-line architectures. Given an evolution for the base of the product-line we would be able to check if it has impact on different previous customisations. The opportunity to capture and analyse the interactions between reuses and evolutions is the major benefit from this approach.

6.2.3 New language features

In a hierarchy of architectural descriptions, the architectural descriptions that are not leaves can be considered as specifications for the 'more refined' ones. In this context some specific features can be added to the language that would make the specification language more expressive. For example, we could add the concepts of optional elements (the same for gates, links or bindings), mandatory elements (gates, links or bindings) and forbidden elements (gates, links or bindings). An optional element is one that may be present in the reuser architectural models, but that can also be absent. A mandatory element has to be present in the reusers. A forbidden element is one that cannot be present in the reusers. Optional, mandatory and forbidden could be implemented as constraints on the nodes and edges that represent elements, gates, links and bindings.

This is important because it provides the means of some 'anticipated evolution', allowing the original designers of the architecture to express or constrain possible adaptations of the architecture.

The preconditions in the increments would ensure that if architectural model A reuses architectural model B then:

- if the element E is optional in B then it can be mandatory or optional, in A, or it can even be absent,
- if E is mandatory in B then it cannot be optional nor forbidden in A,

- if the element E is forbidden in B then it cannot be optional or mandatory in A.

New conflicts would appear due to the implementation of these features in Adela.

6.2.4 Styles

Recall that in section 6.1.3.4 we found a limitation in the representation of styles because it was not possible to express standard system configurations that would later be extended and customised. The ability to express hierarchies of architectural descriptions provides us with this possibility. In this way we can express that a client-server architecture is one in which the elements are clients and servers, communicating through service-requests connectors, and in which there is at least one (mandatory) server. We could reuse this 'style' defining another more complex style, with a (mandatory) name server that communicates with the clients in order to provide access to the server. From this last style the architectures of different systems could be derived, by adapting the architectural description to more specific needs, e.g. defining new clients, more than one server, etc.

6.3 Summary

In this chapter we presented several extensions to the language Adela. We saw that it is possible to extend Adela with more complex operations, in order make architectural descriptions shorter, with different kinds of elements, like components and connectors and even finer classifications (like pipes, filters, clients, servers, etc), and with specific restrictions on the architectures, representing styles. We also saw how to enrich the language with support for reuse, in this way allowing us to find conflicts in the interaction between reuse and evolution of an architectural description.

7 CONCLUSIONS AND FUTURE WORK

Using our initial goal as a starting point, in this chapter we will summarise what we did in order to achieve it. Then we will describe some open questions and possible directions of future work. Finally we will state which are the main contributions of this dissertation.

7.1 Summary

The major goal of this thesis was to show that the reuse contract formalism can be applied to the level of software architectures and provides a good framework to document and reason about unplanned evolution of software architectures. In particular, it is well suited to detect conflicts that can appear during the evolution of a software architecture, and more specifically conflicts caused by the interaction of independent evolutions.

We proved this by constructing an architecture description language to describe software architectures that handles evolution conflicts. The conflict detection capabilities in this language are based on the reuse contracts approach.

The architecture description language is called Adela and it can describe software architectures and express modifications to an architectural description. This language was presented in Chapter 4. An expression in Adela is a sequence of increments, each of which expresses a modification to an architectural model, i.e. a set of architectures. Each increment can be a single modification or a pair of modifications that are done in parallel by different users. The semantics of an expression in Adela is the final description of an architectural model together with the conflicts that were found in the application of the increments. Given an expression in Adela, its semantics is obtained as follows: first the increments are translated into basic reuse contract types, then the reuse contract types are applied to a graph representation of the architectural model, and finally the resulting graph is interpreted as an architectural model. The translation of the increments into reuse contract types is described in chapter 4, as well as the mapping from graphs to architectural models.

We analysed and defined different kinds of conflicts that appear as the unexpected interaction between Adela increments. We did this based on a case study: the original architecture of the SOUL system and its evolutions. We classified the conflicts as applicability (syntactic) conflicts and evolution (semantic) conflicts. We expressed the conflicts in terms of graphs and integrated them into the reuse contract model. This is described in Chapter 5.

We developed a prototype interpreter for Adela expressions in PROLOG. This prototype receives as input an Adela expression and returns a pair containing a description of the resulting architectural model and the conflicts that were generated. The prototype uses the PROLOG implementation for reuse contracts developed by Tom Mens [Mens99].

While experimenting with the PROLOG implementation of Adela we found that the descriptions of architectures of real systems are too long because the increments are very

simple and describe very atomic actions. In order to prove that the approach is powerful enough to support more complex language constructs, we analysed several extensions to Adela. Some of them are inspired by the language constructs provided by other languages in literature, and some others are just syntactical extensions that would help in making Adela's architectural descriptions more compact. These extensions, together with their possible implementations, are explored in the first section of Chapter 6.

One very important problem with Adela is that it does not provide any support for reuse. The reason is that reuse is not supported by the formalism of reuse contracts. But Tom Mens proposed a solution for this limitation of the formalism in his thesis [Mens99]. Based on this solution, a solution for reuse of architectural specification in Adela is proposed in the second section of chapter 6.

7.2 Future work

Some possible directions of future work that appeared as natural extensions of this work are analysed in this section. They are organised in terms of the different topics they address.

7.2.1 Reuse Contracts

The reuse contracts approach proved to be an excellent framework to reason about simultaneous evolution. The concepts involved are very simple, they are very well explained in the documentation and they very soon provide a powerful context in which it is possible to reason about evolution.

One important drawback is that each new contract type that needs to be added interacts with all others, and its interactions and possible conflicts have to be analysed separately. Moreover, there is no abstraction mechanism provided for concepts like conflicts, conditions to satisfy or increments. They are basic primitive entities and modifying them or adding new ones involves changing the RC formalism itself. This is reflected in the implementation. As we could experiment in the implementation of the extensions to Adela proposed in chapter 6, each new feature has to be added by hand to the implementation, thus making extensions not so trivial. In the area of software architectures the extension capabilities of the underlying framework are very important, since many new concepts and abstractions often appear, and they need to be incorporated to the language. Styles are an example.

One possible solution to this limitation is to enhance the language with a constraint language, and allow attaching constraints to nodes and edges. This alternative is also discussed in Mens dissertation [Mens99]. In our particular case, a very simple constraint language that allows expressing node constraints such as 'all the elements nested inside me should be of type X or Y' would enable us to express styles. The meaning and behaviour of the constructs provided by the language of constraints should be defined by the reuse contract framework. The constraint language should be carefully designed. Constraints allow the users of the reuse contracts framework introducing new checks. Applicability conflicts would be raised when the constraints are not met. However, the definition of evolution conflicts based on the specific constrains specified by the user would be more difficult. This does not seem to be a straightforward extension.

7.2.2 Definition of new evolution conflicts

New evolution conflicts can be defined for software architectures. In order to find interesting and useful conflicts it is necessary to have insight in the problems that arise in the evolution of software architectures and this can only be done in the context of a big industrial case study. This case should be developed in a collaborative environment, based on product-line architectures.

7.2.3 Adela as a change manager

Adela could take part in the change manager of a tool for configuration management of architectural descriptions. Since the increments that Adela provides are very simple and general, it is very easy to embed Adela in most of the architecture description languages we found in the literature. Even if only a subset of the features provided by the host language are checked for conflicts this would be of great help and an important step forward in providing support for specification-time evolution.

7.2.4 Runtime evolution

In order to address runtime evolution we would need to enhance the language Adela with behavioural information, e.g the order in which communications between elements take place. Without behavioural information we are not able to ensure that the system will go on running in a safe way during the course of a change.

The enhancement of Adela with behavioural features can be gradual. Initially, Adela could take part in the runtime configuration manager and in this way integrated in a runtime architecture modification language. In the initial phase only structural evolution conflicts would be detected. Gradually Adela could be extended in order to detect conflicts based on behavioural information as well. One interesting experiment would be the application of this technique to one of the architecture modification languages in the literature (like C2's AML).

7.2.5 Reuse of architectural descriptions

Recall the documentation of reuse information in the graph by explicitly representing the reuse of an architectural model with an edge, proposed in section 6.2.1. The preconditions and edge-type constraints for the *reuse* edge define the semantics of reuse. Different semantics for reuse could be explored, each of them expressed with an edge of a different type. For example one could state that an architectural description complies to another if it has the same structure (in that case the names do not matter). Another one can state an architecture complies with another if both the structure and the names are the same, but elements that in the original architecture do not have an internal description can have it in the reuser. A third one can determine that the reuser architecture, in order to be adapted to a specific need, can have more elements and relationships than the original architecture, but never delete one. Many others can be imagined. The different kinds of reuse relationships can even be combined, and their relationships and interaction could be studied in the context of this framework.

We can consider the relationship between an element and the architecture that constitutes its internal description as a refinement relationship, in the sense that the description refines the element. If this is the case, this approach provides us with an environment in which

refinement, reuse and evolution are first-class relationships whose semantics and relationships can be defined, studied and implemented in an homogeneous way.

7.2.6 Adela integrated in a traceability environment

It is important to keep the source code conform to the architectural description of the system, and to propagate the modifications of the architecture to the code, and vice versa in a consistent way.

Kim Mens and Roel Wuyts proposed the use of virtual classifications to represent a mapping from source-level artefacts to higher-level architectural elements, and showed how it is possible to check conformance of the implementation with respect to the architectural description of the system [Mens&Wuyts99]. In this dissertation we studied the interaction of two changes in the level of architectures, but this is not enough. Given an evolution in the architectural level, is the code still compliant? Given a change in the implementation, does it have impact in the architectural level? The same technique used in this dissertation can be applied to detect conflicts between the two levels, i.e. architectural description and implementation. Reuse contracts can be used as a general underlying foundation for documenting the modifications and detecting conflicts.

7.3 Main Contributions

This work has two main contributions:

- It shows RC model is very general and applicable to domains for which it was not originally intended, and in particular, that it is well-suited for the area of software architectures.
- It provides a formalism to deal with simultaneous, independent and unplanned evolutions of software architectures. To our knowledge this has not been done before.

8 REFERENCES

[Allen&Garlan94] Robert Allen & David Garlan: *Formal Connectors.* Technical Report CMU-CS-94-115, Carnegie Mellon University, March 1994.

[Allen&Garlan97] Robert Allen & David Garlan: *A formal basis for architectural connection*. ACM Transactions on Software Engineering, July 1997.

[Codenie&al97] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen: *From Custom Applications to Domain-Specific Frameworks*. Communications of the ACM October 1997, Volume 40, Number 10, Special Issue on Object-Oriented Application Frameworks, pp. 70-77, 1997.

[DeHondt98] Koen De Hondt: *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD Dissertation, Department of Computer Science, Vrije Universiteit Brussel, December 1998.

[Gamma&al94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Garlan&al94] David Garlan, R Allen and J. Ockerbloom: *Exploiting style in architecture design environments*. Proceedings of SIGSOFT'94: Foundations of software engineering, pages 175-188, December, 1994.

[Garlan&Shaw96] David Garlan and M. Shaw: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[IEEE95] IEEE Software Journal, Special Issue on Software Architecture, IEEE Press, November 1995.

[IWPSE98] Proceedings of International Workshop on Principles of Software Evolution, Kyoto, Japan, 1998. ACM SIG Publication, ACM Press, 1999.

[Kramer&Magee98] Jeff Kramer and Jeff Magee: *Analysing Dynamic Change in Software Architectures.* In [IWPSE98].

[Lucas97] Carine Lucas: *Documenting Reuse and Evolution with Reuse Contracts*. PhD Dissertation, Department of Computer Science, Vrije Universiteit Brussel, September 1997.

[Luckham&Vera95] D. C. Luckham and J. Vera: *An event-based architecture definition language*. IEEE Transactions on Software Engineering, pages 717-734, September 1995.

[Magee&Kramer96] J. Magee and J. Krammer: *Dynamic Structure in software architectures*. Proceedings of the ACM SIFSOFT'96: Fourth Symposium on the foundations of software engineering (FSE4), pages 3-14, San Francisco, CA, October 1996.

[Medvidovic&Rosenblum97] Nenad Medvidovic and David S. Rosenblum: *Domains of concern in Software Architectures and Architecture Description Languages* . Proceedings of the 1997 USENIX Conference on Domain-Specific Languages, October 15-17, Santa Barbara, California, 1997.

[Medvidovic&Rosenblum99] Nenad Medvidovic and David S. Rosenblum: *Assessing the Suitability of a Standard Design Method for Modelling Software Architecture.* Proceedings of

Working IFIP Conference on Software Architecture, pp. 161-182, Kluwer Academic Publishers, 1999.

[Medvidovic&Taylor97] Nenad Medvidovic and R. N. Taylor: *A framework for classifying and comparing architecture description languages*. ESEC'97 Proceedings, Zurich, Switserland, pp. 60-67, 1997.

[Medvidovic&al99] Nenad Medvidovic, David S. Rosenblum, R. N. Taylor: *A language and environment for Architecture-based software development and Evolution*. Proceedings of the 21th International Conference on Software Engineering (ICSE 99), Los Angeles, California, May 1999.

[Mens99] Tom Mens: *A formal foundation for Object-Oriented software evolution.* PhD Dissertation, Department of Computer Science, Vrije Universiteit Brussel, June 1999.

[Mens&Wuyts99] Kim Mens and Roel Wuyts: *Declaratively Codifying Software Architectures using Virtual Software Classifications.* In Proceedings TOOLS Europe'99, Nancy, France, February 1999.

[Minsky99] Naftaly H. Minsky: *Towards Architectural Invariants of Evolving Systems*. Submitted to ESEC '99.

[Moriconi&al95] M. Moriconi, X. Qian and R.A. Riemenschneider: *Correct architecture refinement*. IEEE Transactions on Software Engineering, pages 356,372, April 1995.

[Oreizy96] Peyman Oreizy: *Issues in the runtime modification of software architectures*. Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science, University of California, Irvine, 1996.

[Oreizy98] Peyman Oreizy: *Issues in modeling and analyzing dynamic software architectures*. Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis. Marsala, Sicily, Italy. June 30 - July 3, 1998.

[Oreizy&Taylor98] Peyman Oreizy, Richard N. Taylor: *On the role of software architectures in runtime software reconfiguration*. Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4). Annapolis, Maryland, May 4-6, 1998.

[Oreizy&al98] Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor: *Architecture-based runtime software evolution*. Proceeding of the International Conference on Software Engineering 1998, Kyoto, Japan, 1998.

[Perry97] Dewayne E. Perry: *State-of-the-art: Software architecture.* Proceedings of International Conference on Software Engineering, pp. 590-591, ACM Press, 1997.

[Perry&Wolf92] Dewayne E. Perry, Alexander L. Wolf: *Foundations for the study of software architecture.* Software Engineering Notes, vol 17, no 4, October 1992, ACM Sigsoft, 1992.

[Parnas94] D. L. Parnas: *Software Aging*. Proceedings of 16th International Conference on Software Engineering ICSE '94, IEEE Press, 1994.

[Pree97] Wolfgang Pree: *Component-Based Software Development: A New Paradigm in Software Engineering?* Software Concepts & Tools, 18(4): 169-174, Springer-Verlag, 1997.

[Robbins&al98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles and David S. Rosenblum: *Integrating Architecture Description Languages with a Standard Design Method.* ICSE '98 Proceedings, Kyoto, Japan, pp. 209-218, IEEE Press, 1998.

[Shaw&Clements96] Mary Shaw, Paul Clements: *Toward Boxology: Preliminary classification of architectural styles*. Proceedings of the Second International Software Architecture Workshop (ISAW-2), pages 50-54, San Francisco, CS, October 1996, A. L. Wolf ed., 1996.

[Steyaert&al96] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt: *Reuse Contracts: Managing the Evolution of Reusable Assets*. Proceedings of OOPSLA '96, ACM SIGPLAN Notices, 31(10): 268-286, ACM Press, 1996.

[Svalhnberg&Bosch99] Mikael Svahnberg and Jan Bosch: *Characterising the evolution of product line architectures*. Submitted, April 1999.

[Vestal96] S. Vestal: *MetaH Programmer's Manual, Version 1.09*. Technical Report, Honeywell Technology Center, April 1996.

[Wermelinger98] Michel Wermelinger: *Software Architecture and the Chemical Abstract Machine*. In IWPSE98.

[Wermelinger99] Michel Wermelinger and José Luiz Fiadeiro: *Algebraic Software Architecture Reconfiguration*. Submitted to ESEC '99 Conference, 1999.

[Wuyts99] Roel Wuyts: *Declarative reasoning about the structure of object-oriented systems*. In Proceedings TOOLS USA'98, IEEE Computer Society Press, 1998.