VRIJE UNIVERSITEIT BRUSSEL – FACULTY OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCE – PROGRAMMING TECHNOLOGY LAB

A FORMAL FOUNDATION FOR OBJECT-ORIENTED SOFTWARE EVOLUTION

PH. D. DISSERTATION

Tom Mens

August 27, 1999

Supervisor: Prof. Dr. Theo D'Hondt

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I thank Prof. Dr. Theo D’Hondt for supervising this dissertation, and for his support during my doctoral research at the Programming Technology Lab.

I thank Koen De Hondt, Kris De Volder, Carine Lucas, Kim Mens, Marc Van Limberghen, Roel Wuyts and Bart Wouters for proof-reading parts of this dissertation. I especially thank Kris De Volder, who was my most thorough proof-reader. He always tried to understand every smallest detail, thereby giving me invaluable feedback about things that were still missing. Kim Mens was also very helpful, for carefully reviewing and discussing the formal aspects of this dissertation, and for forcing me to make the story more readable. I hope he succeeded. Together with Natalia Romero, he contributed to this dissertation by customising the proposed formal framework to the domain of software architectures. I also thank Veronica Argañaraz for some discussion about the customisation to class diagrams.

I thank all other members of the Programming Technology Lab and the Department of Mathematics, for relieving me of certain tasks during the writing of this dissertation, and for being great colleagues. I also thank the people at MediaGeniX and Wang Global for their practical interest in reuse contracts. In particular I thank Patrick Steyaert, for guiding me during the first few years of my doctoral research.

Special thanks go to Prof. Martin Gogolla for convincing me of the import of graph rewriting in computer science, and for pointing me to the notion of conditional graph rewriting. I also thank Prof. Dirk Janssens for the informal discussion about my dissertation, and for pointing me to the notion of typed graph transformations.

I thank my parents, for giving me the opportunity to study.

I thank my friends, for providing the necessary distraction once in a while, and for putting things in perspective.

Last, but certainly not least, I thank Inge Dedeurwaerdere for her unconditional support and understanding during the final weeks of writing this dissertation, and for wanting to become my wife.

I. THE THESIS

This chapter describes the thesis that will be defended in this dissertation. More specifically, it claims that the principles behind software evolution are domain-independent, and that a formal foundation (based on reuse contracts) should be developed to deal with evolution in a scalable way.

I.1 INTRODUCTION

I 1.1 PROBLEM STATEMENT

In recent years, a lot of effort has been put in trying to make software systems more reusable. Examples are the wide adoption of the object-oriented paradigm (with inheritance and polymorphism as main features to enhance reuse), the acceptance of frameworks, the use of interfaces as opposed to classes, the introduction of all kinds of patterns, the focus on class collaborations rather than isolated classes, the importance of domain analysis, and many more. All these techniques have been shown to enhance the reusability of software components. Nevertheless, in order to create adequate reusable components, *evolution* is crucial, because good reuse can only be achieved after a component has been evolved several times. Indeed, it is inconceivable to predict all possible uses of a reusable component upon its conception. Moreover, since reusable components have a long life span, they keep on evolving after they have been developed and reused.

Unfortunately, there are still many difficulties related to software evolution. Problems with version proliferation, change propagation, architectural erosion, the ripple effect, software entropy, the fragile base class problem, etc. are frequently cited in current literature [IWPSE98]. The plethora of research articles devoted to these problems shows how important these problems are. To cope with these problems, the following *thesis* will be defended in this dissertation:

> *A formal foundation for reuse contracts allows us to deal with software evolution*
>
> *in a domain-independent and scalable way.*

An explicit part of this thesis is the notion of *reuse contracts*. The reuse contracts technique has been shown to help in solving problems related to software evolution by making evolution more disciplined. In [Steyaert&al96], reuse contracts were introduced as a way to deal with unanticipated conflicts in evolving object-oriented class hierarchies. The underlying ideas of reuse contracts are however applicable in a much broader context. For example, in the Ph. D. dissertation of Carine Lucas [Lucas97] it was shown how reuse contracts can deal with reuse and evolution of collaborating classes. [Codenie&al97] indicated that reuse contracts can provide help when evolving a real-world object-oriented application framework. [D'Hondt98] showed how to deal with evolution of software requirements in Object Behaviour Analysis [Rubin&Goldberg92]. Finally, in [DeHondt98], Koen De Hondt has performed some promising experiments with reverse engineering based on the reuse contracts methodology.

Although these results indicate that the ideas of reuse contracts are general enough to provide support for evolution in all phases of the software life-cycle, from requirements specification to implementation, there is still a lot of work to do to validate this claim. Until now, each time the ideas of reuse contracts were applied to a different domain, everything needed to be redefined from scratch:

- What is a reusable/evolvable component?
- How can a component be modified (upon reuse or evolution)?
- What are the possible relationships between components?
- What are the possible conflicts in related components when a component evolves?

While the answers to these questions often are partly specific to the domain to which reuse contracts are applied, we have observed many similarities between all the different domains. For example, similar primitive modification operations recur in each domain, albeit sometimes in a different context. Also, a lot of the associated evolution conflicts arise in many different situations.

By providing a general *formal foundation for reuse contracts*, the conviction can be validated that reuse contracts applied to different domains have a lot in common, and that *the principles behind software evolution are domain-independent.* Although we make use of reuse contracts to illustrate this point, we do not claim that reuse contracts are the only viable alternative for dealing with software evolution in a domain-independent way. We only claim that, based on reuse contracts, a formal framework can be defined which allows us to deal with interesting evolution problems such as analysing the impact of changes, detecting conflicts between parallel evolutions, etc… This formal framework can be customised to different domains by adding domain-specific modification operations and well-formedness constraints. Among others, this allows us to fine-tune the detection of evolution conflicts to specific situations. In this way, the amount of work needed to provide support for reuse and evolution in a new domain is reduced significantly.

A *formal* foundation is beneficial for other reasons as well:

- If software artifacts and their evolutions would be specified in an informal way, e.g., in natural language, many problems can occur. First of all, an informal model is often *incomplete* or *underspecified*, so that tool developers will have to guess or invent the missing specifications. A second problem is that parts of the model can be *ambiguous*, allowing different possible interpretations. Thirdly, parts of the model can be *redundant*, in the sense that the same thing is mentioned many times in different places. When no consistency is maintained between these redundant parts, this can eventually lead to a model that is *inconsistent*, in the sense that different parts of the model contradict each other. Most of these problems can be solved or at least reduced to a large extent by resorting to a formal model.
- Depending on the underlying formalism that is chosen (e.g., category theory, set theory, domain theory), one can make use of existing theorems and results to prove interesting properties like soundness (everything that can be expressed makes sense), completeness (everything that needs to be expressed can be expressed), confluence (all different paths lead to the same result), correctness and consistency, equivalence between different models, etc…
- An underlying formalism also *facilitates tool support*. For example, it can be used for (partial) verification of models, for simulation of models, for automatic code generation, etc.

As a final aspect of this thesis, the *scalability* issue of reuse contracts needs to be addressed. The ideas of reuse contracts should be taken to a higher level of abstraction. This can be achieved in different ways:

- It should be possible to deal with *arbitrarily complex reuse contracts* instead of only primitive ones. In order to do this, the formalism must be powerful enough to derive results from the simple cases to similar results in the general case. This scalability should be achieved at two levels: on the one hand one should be able to deal with arbitrarily complex software components; on the other hand the formalism should allow for arbitrarily large evolution steps.
- Wherever possible, abstraction mechanisms (such as nesting and encapsulation) should be applied to reduce the inevitable complexity in medium-size and large software systems.
- In large systems it is also virtually impossible to understand and maintain the software by looking at the implementation only. In these situations, analysis and design are very useful, as they offer a higher and more abstract view on the software system. Moreover, studies show that significantly more benefits can be gained from reuse during the analysis and design phase than during the implementation phase [Karlssion95]. By defining a formal framework for reuse contracts in a domain-independent way, it will be possible to show that the ideas are applicable to requirements, analysis and design models as well. As a result, the fundamental principles behind reuse and evolution are not only domain-independent, but also independent of any specific phase in the software life-cycle.

I 1.2 Research Restrictions

Since the thesis put forward in the previous section is very ambitious, it is impossible to prove it in its entirety in this dissertation. Therefore, the following research restrictions are made that allow us to focus on the essence of the problem.

We will restrict ourselves to the object-oriented paradigm.

Throughout this dissertation, it is assumed that the reader is familiar with the basic notions and concepts of OO. The fact that we restrict ourselves to *object-oriented* techniques for reuse and evolution doesn't mean that these techniques (such as inheritance, polymorphism, frameworks, patterns, and of course reuse contracts themselves) are not used in, or applicable to, other paradigms as well. However, since many of these techniques were first explored in the object-oriented paradigm, the research is usually more mature in this area. For the interested reader, Paul Bassett [Bassett97a] gives an excellent overview of how to apply the ideas of object-oriented reuse to non-OO languages.

We will use category theory as an underlying formalism.

An important decision that needs to be made is the choice of a formalism to be used as a general formal foundation for reuse contracts. The decision to use *category theory* is based on the following reasons.

- First of all, category theory provides an excellent basis for dealing with *structural relationships*, thus avoiding the need to introduce explicit structuring primitives. This is of particular importance to reuse contracts, which try to detect evolution conflicts by looking at the structure of software rather than looking at its behaviour [Lucas97].
- Secondly, the abstractness of category theory allows us to express all ideas *independent of a specific domain*. Without changing the underlying theory, it can be applied to software specifications, software designs as well as implementations. Indeed, a main objective of the thesis is to define a formalism for reuse contracts in a domain-independent way.
- Category theory also provides powerful and general support for *composition mechanisms*, which can be used to address the scalability issue of reuse contracts, another topic that will be addressed in this dissertation.
- Finally, category theory has a *visual notation*, since it basically reasons about objects and relationships (morphisms) between them. The objects can be regarded as nodes in a graph, while the relationships can be represented as edges in a graph.

Of course, using category theory also has some disadvantages. Because it is a very abstract branch of mathematics, it is sometimes difficult to understand. Therefore, the underlying theory will be hidden from the user as much as possible, by putting a more concrete layer (reuse contracts) on top of the abstract formalism.

Because the formalism of category theory in its entirety is too broad in scope, a further restriction needs to be made. More specifically, we need to decide how reusable software components and their evolution will be represented.

We will use nested labelled typed graphs to represent software components.

The reason why *nested labelled typed graphs* are proposed to represent reusable and evolvable software components is manifold. First of all *graphs* are an intuitive, visually attractive, general and mathematically well-understood formalism. The edges in the graph represent software dependencies, since probably the most important aspect of understanding a software system is understanding the different kinds of dependencies between the different parts of the system. Secondly, a *typing* mechanism is introduced to allow us to distinguish different types of nodes (software components) and edges (software dependencies) with similar characteristics. By making use of type hierarchies, these characteristics can be inherited by subtypes. Finally, a *nesting* mechanism is attached to the graphs to reduce the complexity and to hide unimportant details. Nesting provides an encapsulation mechanism that makes the approach scalable. It also provides an abstraction mechanism, since low-level dependencies between nodes of the graph can be abstracted to higher-level dependencies between the nodes in which they are nested.

We will use conditional graph rewriting
to represent evolution of software components.

In order to represent *evolution* of reusable components, the formalism of *graph rewriting* is chosen. Since graphs are used to represent software components, graph rewriting is a natural choice to represent evolution of these components. Like graphs, the research area of graph rewriting has a solid and large mathematical backing [Gra79, Gra83, Gra87, Gra91, Gra96, Gra98, FI96], while it remains fairly intuitive in use. Additionally, there is a tight connection between category theory and graph rewriting, since category theory is commonly used to define a precise semantics for graph rewriting systems and graph grammars [Ehrig79, Löwe93].

C*onditional* graph rewriting will be used because it is more expressive. With conditional graph rewriting, application conditions are used to determine when a particular production is applicable to a given graph. This is essential to detect evolution conflicts between incompatible evolutions of the same component.

We will express reuse contracts in terms of labelled typed graphs and conditional graph rewriting.

As a final restriction, we will not deal with the research area of software evolution in its entirety, but restrict ourselves to a specific approach which has already proven its use, namely reuse contracts [Steyaert&al96, Lucas97, Codenie&al97, D'Hondt98, DeHondt98, Mens&al99a]. Based on this approach, a domain-independent *framework* will be built on top of the formalisms of labelled typed graphs and conditional graph rewriting. As a result, it becomes possible to use many properties and theorems that have already been proved for graph rewriting.

Although it is possible that other approaches to evolution can also be redefined in a domain-independent fashion, this is outside the scope of this dissertation.

I.2 CONTRIBUTION

This section motivates in more detail why the thesis provides a novel, relevant and important contribution to the object-oriented research community as well as the software evolution community. Even the graph grammar research community might benefit from this work to some extent.

I 2.1 RELEVANCE

When looking at the current state of object-oriented software development, it should be clear that better support for software evolution is indispensable. The lack of adequate mechanisms for software evolution is one of the main causes for the current software crisis. Problems typically arise when upgrading to new versions of software, or when merging parallel evolutions during collaborative software development. At another level, object-oriented analysis and design CASE tools, which are commonly accepted and used to improve the software development process, provide no or poor support for evolution.

This dissertation addresses the lack of disciplined mechanisms for supporting evolution by providing a formal foundation for reuse contracts, which have already shown to be a promising approach to deal with specific evolution problems. Based on the formal foundation, some algorithms will be presented that can be incorporated immediately in CASE tools, so that these tools can provide better automated support for software evolution.

The dissertation is also relevant to the graph rewriting research community, because it provides a practical application of graph rewriting. One of the reasons why graph rewriting is still fairly unknown in the programming community is that from its very beginning in the early 1970's the focus of graph rewriting research was on providing theoretical rather than practical results. As a result, working implementations based on the underlying concepts were not available for a very long time. Fortunately, this is beginning to change. For example, powerful graph-rewriting based visual programming languages like PROGRES [Schürr95] are being developed. Also, the various international workshops that focus on practical applications of graph rewriting indicate a clear shift in interest from more formal research to practically relevant results.

I 2.2 IMPORTANCE

The essential contribution of the thesis is that *the ideas behind reuse contracts are general enough to be applicable in any domain where software evolution is important.* In order to prove this, a formalism needs to be specified that deals with reuse contracts in a *domain-independent* way. At first sight, this may seem an impossible task, because of the wide variety of domains available in software development. For example, dealing with evolution of use case diagrams [Jacobson&al92] does not seem to have much in common with evolution of implementation code.

We will show that a domain-independent formalism for evolution can be found. This is an important result because, in order to add support for evolution to a particular domain, it suffices to customise the domain-independent approach to the specific domain, and all the techniques and formal results for dealing with evolution would be immediately applicable to this domain.

Scalability is also an important issue. Existing work on reuse contracts has shown to be too primitive to be practical. Therefore, this thesis addresses several scalability issues in a formal way, and shows how this can help in making support for software evolution (like, e.g., detecting evolution conflicts and removing redundancy) more efficient.

I 2.3 NOVELTY

The *scalability* aspect of reuse contracts is new, in that existing work on reuse contracts has not investigated its scalability in full detail. [Lucas97] did specify how to deal with composite reuse contracts, but only in an informal way. This dissertation builds further on the work of [Lucas97], and addresses some other scalability issues as well.

According to our detailed literature study, the idea of defining a *domain-independent* formalism for software evolution is also new. All formalisms and techniques for evolution we have encountered in the literature restrict themselves to a specific domain. Most of them are summarised below, categorised according to the domain to which they are applicable:

- Requirement specifications: [D'Hondt98, Ecklund&al96, Wiels&Easterbrook98]
- Implementation level: [Lehman&Belady85, Steyaert&al96, Mezini97]
- Software architectures: [Kramer&Magee98, Wermelinger98, Tokuda&Batory98b]
- Design models: [Bohner96, Lucas97, Mens&al99a]
- Object-oriented software frameworks: [Codenie&al97, Roberts&Johnson96]
- Object-oriented database schemas: [Banerjee&al87, Barbedette91, Bergstein94]

Of all these approaches, only [Wiels&Easterbrook98] seems suitable to be generalised to a domain-independent formalism, since the approach is based on category theory, a formalism which is abstract enough to transfer the underlying ideas relatively easy to new domains.

Sometimes, co-evolution in different domains is dealt with, but even then it is restricted to specific domains. For example, [Katayama98] and [Pirker&al98] discuss co-evolution between specification and implementation, but do not say how these ideas can be generalised to other domains as well.

To summarise, we did not find any approach which profiles itself as being applicable to many different domains. Yet, a domain-independent approach has the important advantage of being more general than a domain-specific one. It allows us to reason about evolution without needing to deal with unnecessary domain-specific details. It also allows us to apply the approach to any domain where evolution is considered important, with significantly less effort than if we would have to start from scratch.

I 2.4 Larger Context

This work is part of a larger research effort aimed at providing support for evolution throughout the software life-cycle. This support consists of a full-fledged methodology based on reuse contracts. It involves theoretical aspects, such as a formal foundation for reuse contracts, as well as practical aspects, such as a set of tools that automate the support for software evolution.

Besides the many research articles that have been published on the subject, the most important research results have been reported upon in several Ph. D. dissertations. This dissertation is the third in a row. First, [Lucas97] established the foundation and terminology for a disciplined methodology for reuse and evolution. Next, [DeHondt98] proposed several tools to automate some of these ideas in an integrated software development environment. This dissertation extends both previous ones, by defining a scalable underlying formalism for reuse contracts, showing its domain-independence, and discussing its impact and relevance for the development of new automated tools that address the issue of software evolution.

I . 3 MOTIVATION

I 3.1 WHY REUSE CONTRACTS?

In order to find similarities between evolution in different kinds of domains, one needs to have an approach that has already been applied to different domains. This is the main reason why we decided upon reuse contracts. Originally, reuse contracts were developed to deal with evolution of classes at implementation level [Steyaert&al96]. Later the focus shifted to design level, by looking at evolution of class collaborations [Lucas97]. Reuse contracts have also been applied to evolution of UML interaction diagrams [Mens&al99a]. Even at the level of requirements analysis, reuse contracts have been applied [D'Hondt98].

For all these different domains, the same approach was followed: first, a number of primitive evolution operations was proposed, and next, potentially conflicting interactions between these primitive operations were identified. Despite the simplicity of this approach, the results were promising. Additionally, the simplicity allowed us to obtain a better insight in the primitive mechanisms for evolving a software artifact, and the possible conflicts that can arise when a software artifact evolves. As it turned out, it was possible to identify a set of primitive modification operations that recur in each of the considered domains, as well as a set of basic evolution conflicts that recur in all these domains. This dissertation takes these basic operations as a starting point, and uses a formal approach to try and find more sophisticated results.

I 3.2 WHY A FORMAL FOUNDATION?

In order to reason about disciplined software evolution, a formal approach should be followed. First of all, it allows us to give a precise and unambiguous definition of the concepts involved in software evolution. E.g., what is a software artifact, what is evolution, what is an evolution conflict? Lack of a formal foundation can also give rise to incompleteness, inconsistency, etc… at different levels.

By restricting evolution to a well-defined set of elementary modification operations, it is possible to give a *complete* characterisation of possible evolution conflicts that may arise when these operations are used to evolve a software artifact.

In order to avoid CASE-tools that support evolution in an ad-hoc fashion, a formal approach is also necessary. This has the additional advantage that it makes the CASE tool easier to understand and maintain, since the underlying ideas are well-defined. In this dissertation several algorithms will be presented (based on our formal framework) that can be used immediately to add evolution support in existing or new tools. Moreover, since the algorithms are expressed in a domain-independent way, they can be applied to every domain where more support for disciplined evolution is required.

I 3.3 WHY NESTED LABELLED TYPED GRAPH REWRITING?

While the benefits of a formal foundation have already been discussed in the introduction of this dissertation, this section motivates the choice of labelled typed graphs and graph rewriting as a formal foundation. The motivation is performed gradually. First, the need for a *visual* formalism is advocated. Next, it is argued that *graphs* are the most appropriate visual formalism for our needs. Then it is explained why a *nesting* and *typing* mechanism is needed. Finally, the use of *graph rewriting* is motivated.

I 3.3.1 Why a Visual Formalism?

When developing a formalism, there are several alternatives. A textual notation can be chosen, a visual one, or a combination of both. From a formal point of view, a textual notation is usually more appropriate because it is more concise. From a pragmatic point of view, a visual formalism is preferred often, since graphical representations are more attractive, intuitive, and easier to understand than pure textual information:

> *"The fields of scientific visualisation and program visualisation have demonstrated repeatedly that the most effective way to present large volumes of data to users is in a continuous visual fashion." [DePauw&al93]*
>
> *"Those programs which rely on a dialog of several objects are much easier to understand with diagrams than just by examining source code." [Cunningham&Beck86]*
>
> *"Our motivation for using conceptual graphs lies in the aim of gaining a better understanding of complex specifications by visualising them. Such an approach is well-tried nowadays and successfully employed by object-oriented methods like OMT or the Coad-Yourdon approach." [Gogolla96]*

There are many different ways in which object-oriented software can be visualised. For example, [DePauw&al93] uses scatter diagrams, histograms and many other kinds of diagrams to focus on different aspects of the behaviour of object-oriented software. Among all these different possibilities, we need to choose the one that is most appropriate for our purpose, namely for dealing with problems related to software evolution. In the next subsection we motivate why *graphs* are the most viable alternative.

I 3.3.2 Why Graphs?

Graphs are based on a well-understood mathematical foundation (graph theory), and encompass a huge number of concepts, methods and algorithms. This makes them very interesting from a formal point of view. From a practical point of view, graphs are also very useful, since they are used often as an underlying representation of arbitrarily complex software artifacts and their interrelationships:

- Graphs have already been proposed by several authors for describing and understanding object-oriented programs, since they provide a compact and expressive representation of program behaviour. One of the earliest proposals was [Cunningham&Beck86], where graphs were introduced to describe the message sending behaviour between objects. This resulted in a better understanding of the Smalltalk-image, and facilitated debugging of object-oriented code. In [Kleyn&Gingrich88] a next step was taken, by using different kinds of graphs to describe the behaviour of large scale object-oriented systems. Besides method invocation graphs, also object invocation graphs, taxonomy (or inheritance) graphs and part-whole graphs were introduced. Each kind of graph presents a different perspective on system behaviour, and each perspective yields different information. In this way, the behaviour of objects can be understood more easily, thus facilitating code sharing and reusability. In [Ellis95] the notion of conceptual graphs was applied to object-oriented concepts.
- Many object-oriented metrics [Chidamber&Kemerer91] are based on a graph (or tree) representation of the object-oriented system. For example, *coupling* is defined as the degree of nodes, and *depth of inheritance* is defined as the longest path in the inheritance graph. Also in [Pfleeger&Bohner90], graph-based metrics are used to evaluate the maintainability of a system whenever a change is proposed.

The above results indicate that graphs are general enough to be used for many different purposes, depending on the interpretation that is given to the nodes and the edges of the graph. Among others, the nodes can represent entities like methods, classes, objects, attributes, packages, components or even entire systems. The edges can be used to represent all kinds of dependencies between the nodes. This is essential, since probably the most important aspect of understanding a software system is understanding the different kinds of dependencies or relations between the different parts of the system. As observed by [DePauw&al93]:

> *Understanding the structure and internal relationships of large class libraries, frameworks, or applications is essential for fulfilling the promise of code reuse.*

An even more important reason why it is important to study dependencies comes from several experiences with conflict detection in independently evolving software components [Steyaert&al96, Lucas97]. Most of the interesting evolution conflicts arise when existing dependencies are inadvertently removed between components, when implicit assumptions are made about particular dependencies, or when particular dependencies between components are implicitly assumed without being actually present.

I 3.3.3 Why Nesting?

> *On the greenboard I drew a state-transition diagram, representing a program. "We should be able to cut any line of the graph and splice a subgraph into the cut automatically," I said to Gunnar. Then I erased a line and drew more circles and lines, in place of the erased line. "Of course, we need to be able to nest the splices, and we should also be able to delete subgraphs as easily as add them." [Bassett97a]*

An essential feature of nested graphs is that the nodes of a graph can contain graphs themselves. This containment relationship is usually referred to as *nesting*. Nesting is a natural way for humans to control the complexity of even a single aspect of a system. It is used as a kind of encapsulation or layering mechanism, by hiding the internal details of a node from other nodes.

Some form of nesting occurs in each phase of the software life-cycle. At implementation level, for example, we find nested methods (in Beta), nested procedures (in Pascal), nested modules (in Modula-2), nested packages (in ADA) and inner classes (in Java). Even object-oriented class hierarchies are a form of nesting. Other areas where we find nesting are: hierarchical data-flow diagrams to describe functionality at analysis level, composite classes in UML, and statecharts [Harel88] or nested state diagrams to model the behaviour of classes. Statecharts allow one to simplify the representation of complex state behaviour through the use of nested states.

I 3.3.4 Why Typing?

Typing of graphs should not be confused with typing in programming languages. In programming languages, typing is used for debugging purposes to make programs safer, and also to increase the readability of programs. With graphs, typing is nothing more than a classification mechanism to distinguish between different kinds of nodes and edges. In this respect, typing corresponds more to the distinction between classes and objects in class-based object-oriented programming languages. All objects that have the same characteristics can be classified in a class that specifies these characteristics. Moreover, an inheritance mechanism on classes can be defined to abstract common characteristics between different classes.

Since classification in combination with an inheritance mechanism has shown to be a very powerful abstraction mechanism in object-oriented programming languages, we have decided to take the same approach with graphs. Each node and edge in a graph has a corresponding type. This type can be used to specify the common characteristics of all nodes (or edges) having this type. Similar to the inheritance mechanism on classes, a subtyping mechanism on types can be used to abstract the common characteristics of types to a common supertype. All characteristics of a particular node or edge type are automatically inherited by all the subtypes.

Again, the above notion of subtyping for graphs should not be confused with the notion of subtyping in object-oriented programming languages. It corresponds more to the notion of subclassing. Unfortunately, some programming languages do not make a distinction between subclassing and subtyping, although these are clearly separate concepts and should be dealt with in an orthogonal way [Cook&al90].

I 3.3.5 Why Graph Rewriting?

Graph grammars or *graph rewriting systems* allow us to describe a possibly infinite collection of graphs in a finite way, by stating a number of initial graphs together with a set of graph production (or rewriting) rules. Through repeated application of these rules starting from one of the initial graphs, new graphs can be generated. The specific form of the graph production rules and the mechanisms stating how and under which conditions a production can be applied to a graph, and what the resulting graph is, depend on the specific graph formalism that is used.

The use of graph grammars (or graph rewriting) in computer science applications is clearly motivated in [Löwe93]:

> *"Graph grammars provide an intuitive description for the manipulation of complex graph-like structures as they occur in databases, operating systems and complex applicative software. Besides that, all approaches to graph transformation systems offer theoretical results which help in the analysis of such systems."*

More importantly, in the area of software evolution, it is natural to represent evolution of software components formally by means of graph rewriting, especially because graphs have been chosen to formally represent arbitrary software components. For object-oriented databases, a transformational approach to describe their evolution was presented in [Banerjee&al87]. This idea was later applied to deal with behaviour-preserving transformations of object-oriented software applications [Opdyke92, Bergstein94, Tokuda&Batory98].

Like graphs, graph rewriting is very intuitive in use, because it can to a large extent be represented visually. Nevertheless, it has a firm theoretical basis, as can be witnessed from the many different theoretical papers appearing on the subject [FI96], and the often recurring international workshops [Gra79, Gra83, Gra87, Gra91, Gra96, Gra98]. These theoretical foundations of graph rewriting can assist in proving correctness and convergence properties of the software. Examples of some interesting properties are: parallel and sequential independence of graph derivations, confluence property, and composition and decomposition of graph derivations. Some of these properties will be needed in this dissertation to prove some specific results related to evolution of software components.

I . 4 THE DISSERTATION

I 4.1 DISSERTATION OVERVIEW

Chapters II and III can be considered as preliminary work, in the sense that they only sketch the context required for proving the thesis.

- Since we restrict ourselves to the object-oriented paradigm, in chapter II we briefly review the ideas of Object-Oriented Software Engineering (OOSE) in general, and look at object-oriented analysis and design methodologies (OOA/D) in some more detail. Next we situate software reuse and evolution in the software life-cycle, and discuss how reuse contracts fit in. Moreover, a discussion of related work is given to illustrate the similarities and differences between reuse contracts and other approaches to evolution.
- In chapter III we present the formalism of labelled typed graphs and conditional graph rewriting. Most of the definitions presented here can also be found in other work, although obviously some definitions will be fine-tuned to suit our specific needs. Readers that are only interested in the intuitive ideas behind this dissertation may skip this chapter.

After this preliminary work, we start with the actual dissertation. The following three chapters each cover a specific aspect of the thesis: a formal foundation for reuse contracts, the scalability of this formalism, and finally the domain-independence of the formalism.

- First, chapter IV shows how a *formal foundation for reuse contracts* can be defined on top of the underlying formalism of conditional graph rewriting. To this end, software components are represented as labelled typed graphs, reuse contract types are defined as elementary graph rewriting productions, and a number of interesting properties are proven about them. The most important part deals with how conditional graph rewriting can be used to detect evolution conflicts between incompatible components.
- Chapter V deals with the *scalability* of the proposed formalism. This is needed in order for the approach to be applicable in practical situations. Issues such as composite contract types, transitive closure, nesting, subtyping and a normalisation algorithm to remove the redundancy in an evolution sequence are discussed.
- In Chapter VI we validate whether the proposed formalism really is domain-independent. First, it is shown that the formalism is a generalisation of existing work on reuse contracts [Lucas97]. Next we show how support for evolution of different kinds of UML diagrams can be expressed. Finally, we motivate that support for evolution can be added to other domains, such as software architectures and object-oriented database schemas as well. In this way, we illustrate that the formalism can be used to add support for evolution during the early phases of the software life-cycle, or even in any domain where software evolution is important.

Finally, Chapter VII summarises the main contributions of this dissertation, discusses some future work, and concludes.

I 4.2 LAYERED APPROACH

To prove the thesis, a layered approach will be taken. More specifically, the formal framework for evolution that will be developed in this dissertation is schematically represented in Figure 1. It is composed of three layers.

The lowest layer is the *underlying formal foundation*. Using category theory, *labelled typed graphs* are defined as objects in a specific category, and *conditional graph rewriting* can be defined in terms of the category-theoretical notions of morphisms and pushouts. A very short introduction to *category theory* can be found in the appendix (chapter VIII).

Layer two, which is built on top of the previous layer, defines a *domain-independent framework for evolution* in terms of these graphs and graph rewriting. Because we have chosen for the reuse contracts approach, we first define *primitive reuse contracts*. Next, to make the approach more scalable, *composite reuse contracts* are defined in terms of these primitive ones.

As a final layer, *domain-specific customisations* can be defined on top of the domain-independent formal framework. More specifically, we will consider several customisations, such as evolving *class collaborations*, evolving *UML class diagrams*, and evolving *software architectures*.

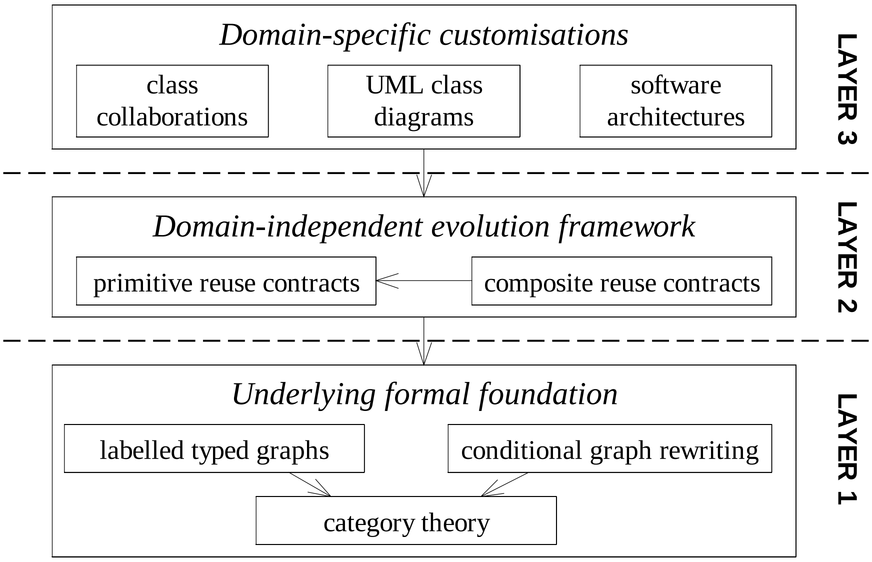

Domain-specific customisations
class collaborations
UML class diagrams
software architectures
LAYER 3
Domain-independent evolution framework
primitive reuse contracts
composite reuse contracts
LAYER 2
Underlying formal foundation
labelled typed graphs
conditional graph rewriting
category theory
LAYER 1

Figure 1: Layered approach of the dissertation

II. Object-Oriented Software Engineering

This chapter provides some intuitive background of object-oriented software engineering, and discusses how software reuse, software evolution and reuse contracts fit into this.

II . 1 INTRODUCTION

This introductory chapter discusses the various issues involved in software engineering, explains how they relate to the ideas of reuse and evolution, and identifies where reuse contracts fit in. The main goal of this chapter is to provide some necessary background for the main topic of this dissertation, namely dealing with evolution in a uniform way throughout the software development process.

Section II . 2 looks in more detail at the different software development phases in general, and then focuses on *object-oriented* software development in particular. Indeed, as a first research restriction in section I 1.2 of the previous chapter, we decided to restrict ourselves to the object-oriented paradigm. Next we focus on the analysis and design phase, because these phases are less mature than the implementation phase. Consequently methodologies and tools that support reuse and evolution there are not as well developed and understood as in the implementation phase.

Section II . 3 takes a detailed look at technical difficulties involved in reuse, such as the proliferation of versions. The next section focuses on evolution aspects, relates it to software reuse, and discusses some graph-based and other approaches to software evolution. Finally, section II . 5 explains the ideas behind reuse contracts, and discusses how they can be used for documenting reuse and evolution in a disciplined way. In this way, reuse contracts can provide better support for change propagation and impact analysis. The relation to other techniques and approaches is explained as well.

II . 2 THE SOFTWARE DEVELOPMENT PROCESS

II 2.1 PHASES IN THE SOFTWARE LIFE-CYCLE

To be able to explain where reuse and evolution fit in the software development process, the software development life-cycle needs to be reviewed. Usually, this life-cycle is subdivided in different phases.

During the *requirements* phase, the requirements for a software system are discovered, specified and analysed. In this sense, this phase includes the so-called *analysis phase* as a subphase.

In the *design phase*, the software system is designed, but still independent of a specific programming language. Several subphases can be distinguished, such as architectural (or high-level) design, mechanistic design and detailed (or low-level) design.

During the *implementation phase*, the actual code is written, based on the information given in the detailed design. Usually, template code can be generated directly from the design. In that case, only the holes need to be filled in.

The *testing and validation phases* check if the software system fulfils the specified requirements, and see if the software behaves correctly in all situations.

Finally, the *maintenance phase* deals with the software system after it has been delivered, by making bug fixes, implementing new requirements, etc… In the context of software evolution, this is a very important phase, since it is the place where software evolution occurs continuously.

We will now take a closer look at most of the phases mentioned above.

II 2.1.1 Requirements

The *requirements phase* is the phase in the software life-cycle where is defined *what* the system should do, rather than how this behaviour should be achieved. Many different terms are used to specify parts of the requirements phase: requirements capture, requirements gathering, requirements elicitation, requirements specification and requirements analysis. However, it is not our intention to look in detail on the (sometimes only minor) distinctions between all these terms.

During the requirements phase, the software engineer tries to discover and formalise the exact requirements (functional as well as non-functional) for the software system. This is necessary to understand what the software system needs to do, and how the system can be used. Functional requirements can be expressed by having informal interviews with the customers or end-users of the system, or by letting the user specify different scenarios of use. Non-functional requirements relate to the quality of a software system (with respect to extensibility, reusability, portability, performance, ease of use, etc.), and are typically much more difficult to express than functional requirements.

Often visual notations like use cases [Jacobson&al92] or message sequence charts [Rudolph&al96] (or the related event traces [Rumbaugh&al91] or sequence diagrams [OMG97a]) are used to specify functional requirements. Another way of making the requirements more precise is by using formal specification languages like OBA [Rubin&Goldberg92], VDM [Jones90], Z [Spivey89] or its object-oriented equivalent Object-Z [Carrington&al90, Duke&al91].

A more intuitive and highly informal approach to capturing requirements is the use of Class-Responsibility-Collaboration (CRC) cards [Wirfs-Brock&Wilkerson89, Wirfs-Brock&al90]. They provide a simple, easy to explain, low-tech approach to working with users to define the requirements for an application. Because CRC models approach requirements from a different angle than do use cases, they are used often to validate the information gathered by use cases (and vice versa).

As another alternative, a prototype (or mock-up) of the user interface of the system can be built, so that users can get a better idea of what the system will look like. In order to avoid the technical aspects involved in building a prototype, one can also make use of interface-flow diagrams [Page-Jones95] that show the relationship between the user interface components, screens and reports that make up the application. These interface-flow diagrams allow one to easily gain a high-level overview of the application interface.

II 2.1.2 Design

If the requirements have been (partially) captured and analysed, the design can start. Before making such a design however, one should first decide on an adequate *software architecture*. To deal with the different aspects of design, the design phase can be subdivided in different subphases. In the object-oriented paradigm, [Douglass98b] proposes to make a subdivision in the following phases:

Architectural design, which describes the software architecture. Software architectures [IEEE95, Shaw&Garlan96] serve primarily as “the big picture” of the system under development. They are the structural and behavioural frameworks on which all other aspects of the system depend, and can be compared with the load-bearing frames in buildings. The software architecture is usually defined as the organisational structure of a software system including components, connectors, constraints, and rationale. This architecture should be chosen well, since changes to it usually require complex and costly changes to substantial parts of the system. A carefully chosen software architecture can greatly enhance the quality of the software system under consideration. Note that in [Jacobson&al97b] the architectural design phase is referred to as *robustness analysis*: “the [robustness] analysis model does not deal with low-level details of the implementation, but concentrates on the high-level static structure of the system, which constitutes the first step towards the system architecture”. Numerous *architecture description languages* have been defined for the purpose of expressing architectural designs more easily [Garlan&Shaw96, Medvidovic&Taylor97].

Mechanistic design, which describes the important object collaborations, usually by means of message interactions. This design serves as a high level description of the software, describing its key features.

Detailed design, which describes the objects internal details such as instance variables and methods. This design gives a blueprint of how the code is organised. Usually, one makes use of class diagrams and state-transition diagrams to describe the detailed design.

II 2.1.3 Testing and Validation

During this phase, the software system is thoroughly tested to see if it behaves correctly in every imaginable situation, and if it fulfils the specified requirements. Defects can be uncovered by executing specialised test programs and test cases.

Testing can and should be performed in all phases of the software life-cycle. The sooner errors or inconsistencies are detected, the better. If inconsistencies in the requirements are only detected during the implementation phase, this can lead to severe delays.

Depending on the CASE-tool that is used, testing can be facilitated or automated to a large extent. For example, in CASE-tools dedicated to real-time systems, like SDL-based tools [Olsen&al94], ObjecTime [Selic&al94] and Statemate [Harel&Gery96], scenarios represented by means of interaction diagrams or message sequence charts can be executed automatically, which greatly facilitates testing. Due to the lack of a precise formal semantics for UML [OMG97b, Breu&al97], CASE-tools for UML do not yet achieve the same amount of automation.

II 2.1.4 Maintenance

A software system is never completely finished. Even after it has been delivered, the software continues to evolve. This process of modifying software after it has been delivered is referred to as *software maintenance*. There are many reasons why software needs to be maintained continuously:

- the requirements can change over time, in response to unavoidable and unforeseeable changes in the real world, or in other systems with which the software interacts
- reported errors may need to be fixed
- by actually using the delivered system, users might come up with new functionalities that are desirable as well, but that were not present in the original functional requirements. This is sometimes referred to as the *Heisenberg principle of software development*: by using a particular piece of software, the perception of the users changes, and they start to see new things for which the software could be used.
- the system needs to cope with the constantly evolving software technology: new kinds of media, new kinds of hardware, new software standards
- to increase the performance of the software
- to enhance the interoperability of the software

The continuous process of maintenance or evolution is typical for software, and characterises the main feature of *soft*ware, namely its modifiability, its capacity for change, its softness. Moreover, the maintenance phase can be considered as the most important phase of the software development process, since studies have shown that the costs of system maintenance (i.e., evolution) are as high as 60% of the overall development costs [Ghezzi&al91]. This is one of the main reasons why more disciplined support for evolution is needed, and why exactly this issue is addressed in this dissertation.

II 2.2 Object-Oriented Software Development

This section focusses on object-oriented software development, because this was one of the research restrictions we made in section I 1.2. As will be seen, the import of evolution becomes even greater when developing object-oriented software, because of the iterative software development style which is usually applied.

II 2.2.1 Introduction

The paradigm of structured programming, invented almost 40 years ago (with the introduction of Algol-60), constituted a major advance in software engineering. Most notably, the idea of information hiding, due to Parnas [Parnas72], led to the notion of abstract data types (ADTs) in module-oriented languages like Modula-2. Despite these promising features, many well-structured programs remained difficult to modify and reuse in a reliable way. The reason for this is that there are many different ways to structure a given program. When the program evolves over time, the chosen structure often becomes inadequate or inefficient to deal with the changed requirements.

In order to cope with this problem, the **object-oriented** paradigm introduced the idea of *programming by difference* or *incremental modification*. A so-called *inheritance mechanism* allows us to adapt existing components (usually classes or objects) without having to edit source code. Another major contribution of object-oriented programming was the introduction of *polymorphism*, which also had a great impact on the reusability of programs. Polymorphism allowed to step away from the separation of code and data: with polymorphism it becomes possible to differentiate over the code that is executed based on the data that is present, without the need for conditional statements.

Although these are definitively steps in the right direction, the object-oriented paradigm still has some shortcomings. For example, when considering the inheritance mechanism, it usually only allows one to modify components by *adding* new features to an existing component or by *modifying* existing features. In some cases, however, particular features need to be *removed* from an existing component. The reason why this is not possible in most inheritance mechanisms is because the child (i.e., the modified component) is required to be *substitutable* for the parent (i.e., the original component). There is also some discussion about which kind of inheritance is more appropriate. Sometimes *implementation inheritance* is advocated, referring to inheritance of code fragments. In other cases, *interface inheritance* is preferred, referring to inheritance of interface specifications. Substitutability can be defined in both variants, although it is not a prerequisite.

II 2.2.2 Iterative and Incremental Software Development

An often employed approach towards object-oriented software development is the process of *iterative and incremental software development*. Instead of developing systems in a linear fashion, where each phase must be completely finished before the next phase can start (as in the traditional waterfall life-cycle), most object-oriented software systems are developed *incrementally*. This means that the software system is developed step by step as a succession of different increments or versions. Within each increment, the software developer *iterates* over the various phases. Usually, the most important requirements (i.e., those with the highest risk) are dealt with first. In later increments the other requirements are added, hopefully in such a way that only small changes are needed to the rest of the system. For this reason, the development of the first few increments is a very important activity, since it is then that we establish an architectural base that must last for the system’s entire lifetime.

II 2.2.3 Software Evolution

In the view of iterative software development, software maintenance can be seen as the life of the software after its initial development cycle (or after the first delivery of the software system). This comprises all subsequent evolutions of the software, such as bug fixes and adding new functionality. It is obvious that, because of this iterative approach, it becomes essential to deal with evolution in a disciplined way. To stress the intrinsic evolutionary nature of software, the term *evolutionary development* is used often instead of iterative development. For the same reason, the term *software evolution* is preferred over software maintenance. Indeed, the term software maintenance does not make much sense when there is no essential difference with the earlier phases.

Although it is necessary to deal with the process of constant evolution, the other side of the coin is that upon each evolution step the software complexity (or *software entropy*) increases, and the software becomes progressively less useful if no proper actions are taken. Indeed, software systems tend to evolve in irreversible ways: changes destroy information about the previous version of the system. This problem is referred to as the problem of *software ageing*. As stated nicely by Parnas:

> *"Programs, like people, get old. We can't prevent ageing, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable." [Parnas94]*

If one does not take care, the software drifts away from the original architecture, eventually leading to legacy systems that become very difficult to maintain and adapt. To reduce the ever increasing complexity of a software system, the software needs to be *restructured, redesigned or refactored* at regular intervals [Opdyke92, Opdyke&Johnson93, Johnson&Opdyke93, Tokuda&Batory98b].

Section II . 4 will look in more detail at software evolution, and see how techniques like *impact analysis* can reduce or solve some of the problems related to software change.

II 2.3 Overview of Analysis & Design Models

II 2.3.1 Need for More Abstraction

In software engineering there is a constant tendency to describe systems in an increasingly more abstract and high-level way, to cope with the fact that software systems become more and more complex. Initially, assembly code was used to implement systems. After a while, pseudocode was introduced, which lead to the first "real" programming languages. These languages evolved into today's third generation languages such as Pascal, Modula, Java, C++, Smalltalk, Eiffel, etc. Each of these steps in the evolution of software development environments allowed programmers to describe software at a higher-level of abstraction. This process didn't stop at the level of third generation programming languages. On the contrary, it gave rise to many different analysis and design methods and notations. These allowed software developers to describe software systems in a visual and more intuitive manner. The same observation has been made in [Barbier&al98]:

> *"One of the problems with constructing working prototypes of high-level system models using traditional programming languages is that general-purpose languages do not directly support high-level abstractions (e.g., state machines). This means that a considerable amount of implementation-level effort may have to be invested to construct the necessary abstractions out of the more primitive facilities provided by a programming language. To make matters worse, the execution of the model is difficult to observe because the working model is formulated through a low-level textual formalism."*

Analysis and design methodologies were first developed for structured programming languages. In the object-oriented programming paradigm, the same evolution took place with a delay of 10 years, for the obvious reason that object-oriented programming languages are about a decade younger than their structured equivalents. The result of this is that object-oriented analysis and design methodologies have only recently matured.

II 2.3.2 Structured Analysis and Design Techniques

In the late seventies, people started to think of how procedural programming concepts could be lifted up to the modelling level. In the beginning only *data-centred approaches* were proposed, like entity-relationship diagrams [Chen76, Nijssen&Halpin89], data-flow diagrams or flow charts [Gane&Sarson78] and action diagrams. Their main contribution was that they presented data in a more visible and understandable form than via procedures. Moreover, they usually have a strong theoretical background, giving rise to formalisms such as relational algebra and techniques such as "normalisation", which can be used as a "proof of correctness" of a design.

In the early eighties, data-centred approaches were supplemented with *behaviour-centred approaches*, mostly based on a variant of state-transition diagrams or finite state machines. Another important topic during this period was the introduction of formal specification languages like Z [Spivey89], VDM [Jones90], LOTOS, SDL-88 and Petri nets.

II 2.3.3 Object-Oriented Analysis and Design Techniques

Because of the many problems that were encountered when using a combination of data-centred and behaviour-centred approaches towards structured analysis and design, people started to feel a need to lift up object-oriented programming concepts to a modelling level. As a result, in the late eighties and early nineties, the evolution from programming to analysis and design also took place in the object-oriented programming community. This gave rise to a plethora of different methods, all with slightly differing notations. The main difference with the structured analysis approaches is that the OO methods combined data-centred and behaviour-centred approaches into one method. This made them more intuitive and powerful than structural modelling methods, because of the strong cohesion between data and behaviour.

Inside the wide spectrum of different methods, one can distinguish the informal ones and the more formal ones. The most widespread informal methods are Schlaer/Mellor [Schlaer&Mellor92], Coad/Yourdon [Coad&Yourdon91a, Coad&Yourdon91b], Booch [Booch94], Odell [Martin&Odell95] and OMT [Rumbaugh&al91]. One the more formal side, there are approaches like Syntropy [Cook&Daniels94], statecharts [Harel88], message sequence charts, pre- and postconditions and class invariants [Meyer92].

At the same time, object-oriented variants of already existing formal specification languages were developed, leading to object-oriented specification languages like Object-Z [Carrington&al90, Duke&al91] and SDL-92 [Olsen&al94].

Last but not least, many object-oriented techniques for capturing the requirements of a software system have been developed. Among others, the use of CRC cards [Wirfs-Brock&Wilkerson89], Object Behaviour Analysis [Rubin&Goldberg92] and use cases [Jacobson&al92] can be distinguished.

In late 1997, most of the methods mentioned above have converged to a standard notation, the Unified Modelling Language [OMG97a], which has been accepted as an industry standard by the Object Management Group (OMG). Besides being a standard, the UML is also open, in the sense that new features can be added to it quite easily.

II 2.3.4 Executable Analysis and Design Models

Despite the abundance of different object-oriented analysis and design methods, only in a couple of cases the modelling languages are defined sufficiently well to allow the “interpretation” and “compilation” of high-level models, i.e., full model execution and code synthesis. These distinguished cases are:

- Tools based on the SDL standard [Olsen&al94], sold by vendors like Verilog and Telelogic
- ObjecTime, based on the ROOM method [Selic&al94]
- Rhapsody, based on [Harel&Gery96]

Although these tools are mainly used in the real-time and embedded systems community, they have been shown to reduce development time significantly, and to increase product reliability, when compared to traditional development models. An additional advantage is that it is possible to perform a partial verification of the analysis or design, even when this design is not fully completed yet.

Mainstream and commonly accepted methodologies like UML are still relatively immature in this respect, in the sense that they have no complete formal semantics, and consequently do not allow full code generation. However, given the wide adoption of UML, and the need for more formality to deal with the ever increasing software complexity, this is likely to change in the near future. Some work has already been performed in this direction [Douglass98b].

Recent tools that extend UML with real-time functionality and full model execution are:

- Rational Rose RealTime, which combines the underlying execution machinery of ObjecTime with the user interface of the Rational Rose CASE tool.
- Tools that support the SDL 2000 standard, which combines the UML notation with the SDL real-time specification language.

II . 3 SOFTWARE REUSE

Having given an overview of the software development process in the previous section, we now explain how reuse fits in this process, and what are the important issues and problems related to software reuse.

II 3.1 WHAT IS REUSE?

Many different definitions of reuse exist. According to the Software Productivity Consortium [SPC93],

> *reuse is the use of an asset in the solution of different problems or different versions of a problem.*

In a similar vein, Jacobson [Jacobson&al97b] defines reuse as the

> *further use or repeated use of an artifact. Typically, software artifacts are designed for use outside of their original context to create new systems.*

Alternatively, Paul Bassett [Bassett97a] defines reuse as

> *the process of* ***adapting*** *generalised components to various contexts of use.*

The main difference between the latter definition and the former two is that simply using an asset or artifact in a different context is not considered to be reuse by Bassett. He argues that reusing a component without needing to make any modifications to it can hardly be called reuse; it is simply a *use* of the component. Only when the original component needs to be *adapted* (or modified) before it can be used in the new context, one can speak of true reuse.

Nevertheless, this dissertation adopts the former definition of reuse (i.e. *repeated use of an artifact in different situations, with or without making adaptations to it*) since the majority of the reuse community agrees on this definition.

II 3.2 BENEFITS OF REUSE

Reuse per se is not necessarily desired. Reuse is not a goal in itself, but *a means to an end*. There are many goals that can be achieved by making software more reusable, but they are all meant to lead to a competitive advantage for those companies and institutions involved in reuse. The goals can be subdivided into economic and quality benefits.

Economic benefits are:

- The *productivity* can be increased by reducing the amount of software that needs to be developed. As a result, the *cost* of software development is reduced.
- *Time to market* can be reduced because development time decreases. As a direct result, more applications can be written in shorter time, and the *revenue* increases.
- The *number of resources* required to create a particular piece of software can be reduced substantially.
- The *diversity* of products available to customers can be increased.

Quality benefits are:

- In safety-critical systems, the *risk* can be reduced by minimising the amount of new software that must be developed.
- The software will become *more complete*, since more time can be spent on trying to meet user requirements.
- The *reliability* of code can be increased (i.e., the number of errors can be reduced) by repetitive use of software assets. As a result of this, testing, validation, debugging and maintenance efforts can be reduced.
- The overall software (or product) *quality* can be improved: products can be made more usable, customisable, etc. As a direct result, a software manufacturer will become recognised as a supplier of high-quality products.
- Standardisation and interoperability of products can be enhanced.

There is one other benefit which is more difficult to categorise in the above. By always reusing pieces of software in the same application domain, the knowledge of this domain is improved. Moreover, by explicitating this knowledge in a framework or reuse repository, this knowledge is retained and leveraged.

II 3.3 Problems with Software Reuse

Despite all these benefits, software reuse is no silver bullet. There are many problems related to software reuse cited in the research literature. While [Goldberg&Rubin95] gives an excellent overview of the managerial difficulties involved in software reuse, we will focus on some more technical difficulties here:

- Designing and constructing reusable components requires additional effort estimated from 5 to 10 times the effort required to build non-reusable software components. This goes hand in hand with the fact that a substantial initial investment needs to be made, and it takes a while before a return on investment can be achieved.
- A necessary prerequisite for dealing with reuse is the presence of a *reuse repository*, or library of reusable assets, that can be used for storing and retrieving reusable elements. However, it is not always simple to find a suitable reusable component in the repository, and to understand its relevance. Especially when the component libraries are big it is very difficult to find a suitable component. One first needs to specify which kind of component is needed, look in the library for components that are similar enough, and make some modifications to the selected component afterwards. To be able to do this, however, a means is needed to measure when two assets are *similar enough* to be used interchangeably. One alternative is to use knowledge-based library systems for this purpose [Wood&Sommerville88]. Another alternative is to use a *faceted scheme* to find similarities between existing program components and the desired component [Prieto-Diaz&Freeman87, Liao&al99].
- The “not invented here syndrome” refers to the psychological aspect that programmers often do not trust software written by others, and consequently are reluctant to reuse this software. When you decide to reuse a component made by someone else, who will be blamed if the software doesn’t work?
- In order to achieve systematic reuse, *non-functional requirements* need to be considered in addition to functional requirements. These non-functional requirements are often used to express the *quality* of a software system. They can be subdivided in *reuse-related* non-functional requirements (such as adaptability, extendibility, reusability and robustness) and other more conventional non-functional requirements (such as performance, compatibility, portability, reliability, ease of use and timeliness). However, it is far from clear how the reuse-related non-functional requirements can be achieved and measured.

Adele Goldberg [Goldberg98] raises some other fundamental questions that need to be solved when dealing with reusable components:

- Who is responsible for correcting defects in reusable components?
- When can evolutionary demands permit backward compatibility to be broken, if ever? Usually this is the case if the benefits of the improved components outweigh the additional cost of updating all applications that make use of this component.
- How will one keep track of reusers, or are the reusers expected to keep track of themselves? If no track is kept of reusers, in one way or another, the problem of *version proliferation* will occur. This problem already occurs in traditional version management [Conradi&Westfechtel98], where the different versions of a component evolve at a single source. However, in a free market version proliferation becomes even more severe, since the evolution of versions is more complex and management of version numbers can become a problem itself. If a reuse repository is present, it should be tightly integrated with the version management system, among others by providing a locking mechanism so that reusable components cannot be modified by different reusers at the same time.

Another problem is *overfeaturing*. Reusable components are maintained and modified according to the needs of different users. Of course, not all user requests for changes can be granted, because this would quickly give rise to an overwhelming number of features that need to be implemented. Therefore a careful selection should be made of which user requests should be handled and which should not.

The reuse contracts methodology, explained in more detail in section II . 5 , provides solutions to some of the problems mentioned above. A more general lesson that can be learned from this discussion is that reuse is intimately related to evolution. Therefore, the many issues involved in software evolution are discussed in section II . 4 .

II 3.4 SOFTWARE FRAMEWORKS

If one intends to develop many different applications within the same problem domain, *domain analysis* provides an effective way to improve the adaptability and reusability of the software. According to [Prieto-Diaz90], domain analysis is

> *"a process by which information used in developing software systems is identified, captured and organised with the purpose of making it reusable when creating new systems."*

While traditional application analysis focuses only on the application under development, domain analysis considers several existing and potential applications, and looks at the commonalities and variabilities between them. The result is a *software architecture* or *software framework* that supports the development of several applications in the domain.

Essentially, a software framework represents a configurable design of all or part of a software system. In the object-oriented paradigm, a so-called *object-oriented application framework* describes how a collection of objects work together. This is usually achieved by defining abstract classes which will be subclassed when the framework is applied, and by describing the collaborations between class instances. In order to use the framework actually, one customises (read: configures) it by filling in the variable parts, usually by making concrete subclasses of abstract classes in the framework.

Frameworks reduce the cost of developing an application because they let you reuse design. On the other hand, the cost of developing the framework itself is high. It must be simple enough to be learned, yet it must be directly usable, and it should provide support for features that are likely to change.

According to Wolfgang Pree [Pree97], frameworks will remain the long-term players towards reaching the goal of developing software with a building-block approach. In this sense, frameworks form a specific instance of the more general idea of component-based development, that will be discussed in the next subsection.

In [Bassett97a], Paul Bassett generalises the ideas behind frameworks, so that they can be applied to non object-oriented languages as well. With his so-called *frame-based reuse,* he introduces *frames* as *generic* classes equipped with an active adaptation mechanism that is more powerful and expressive than the standard inheritance mechanism as found in most object-oriented languages. The essence is that frames merge the IS-A relationship (corresponding to the standard inheritance mechanism) with a HAS-A (or part-whole) relationship. As a result, many of the change propagation and version proliferation problems can be dealt with.

II 3.5 COMPONENT-BASED DEVELOPMENT

With component-based development [Szyperski98], software is not built from scratch, but rather by making use of existing *reusable components*. In many aspects, this requires a new way of thinking.

The *developer of reusable components* should decide which parts of a software system are valid candidates to become reusable components. Depending on the situation or software project, reusable components can be class specifications, class implementations, objects, algorithms, entire applications, analysis and design models, patterns, sets of interacting objects, software frameworks, and many more. A necessary prerequisite is that a reusable component should have a well-designed interface and documentation, and that it is general enough to be reusable in different situations. Even then, it takes some time before a component becomes truly reusable. Reusable components have a long life span, because good reuse can only be achieved after a component has been reused and adapted several times [Johnson&Foote88], and because it is simply inconceivable to predict all possible uses of a component upon its conception.

The *maintainer of reusable components* should always keep in mind that the code she is maintaining is likely to be reused in many different applications. Because each application may have slightly different requirements, component modifications may not work for all applications.

The *(re)user of reusable components* needs to cope with the possibility that reusable components evolve. In that case the applications that make use of these reusable components might need to be updated as well.

II . 4 SOFTWARE EVOLUTION

Like software reuse, software evolution is an essential part of the software development process. Moreover, software reuse and software evolution are intimately related. This section discusses the important issues and problems in software evolution. The research that is taking place in this area is also discussed.

II 4.1 PROBLEMS WITH SOFTWARE EVOLUTION

Nearly all software inevitably undergoes changes during its lifetime. Changes can be large or small, simple or complex, important or trivial - all of which influence the effort needed to implement the changes. Experience over the last 30 years has shown that making software changes without visibility into their effects can lead to poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and the premature retirement of the software system. The immaturity of current-day software evolution is clearly stated in the foreword of the international workshop on principles of software evolution [IWPSE98]:

> *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****.*

Some of the important problems that arise because of poor software evolution are:

- Upon each evolution step the amount of disorder in a software system (also known as *software entropy*) increases, and the software becomes progressively less useful if no proper action is taken. Software systems tend to evolve in irreversible ways: changes destroy information about the previous version of the system. By continually making small changes, the original system design becomes distorted. Consequently, the software becomes more difficult to understand, making further changes progressively more difficult. This problem is also referred to as *software ageing*. To minimise the build-up of entropy in software, the changes should be made reversible.
- A related issue is the so-called *ripple effect*, or the issue of *change propagation*. When a given software artifact (which can be any result of an activity in the software life-cycle such as a requirements specification, an architecture model, a design specification, documentation, source code and test scripts) changes, all artifacts that depend on it might require changes as well. In order to find out the potential effect of changes, the technique of *impact analysis* [Bohner&Arnold96a] is needed. If encapsulation and information hiding techniques have been used to make the software highly modular, then the anticipated changes should be relatively localised and have few ripple-effects to other non-related components.
- It is not always advisable to replace a software artifact by its evolved version, because the evolution might give rise to undesired interactions with its dependent artifacts. For this reason, all dependent artifacts should be protected from these incompatibilities. This can be achieved by resorting to a *version management mechanism*. If incompatibilities between an evolved artifact and a dependent artifact are detected, the old version of the evolved artifact will be used. Of course, one should take care that this does not lead to a *proliferation of versions*. Once in a while, artifacts that still work together with old versions of other artifacts, will need to be upgraded, even if this requires significant changes. Otherwise the software is likely to turn into a *legacy system*, which is exactly what evolution intends to avoid in the first place. In [Conradi&Westfechtel98] an excellent overview is given of the various version management tools that are currently available, commercially as well as research prototypes. It classifies the different versioning paradigms and defines and relates their fundamental concepts.
- Each time a software component is changed, its corresponding documentation should be updated accordingly. Due to time pressure this is often neglected, leading to *inconsistent documentation*.

Software reuse is intimately related to software evolution. All problems related to software evolution are even more important in the case of reusable software artifacts, for a number of reasons:

- Reusable artifacts typically have a long life span, and are hence more subject to evolution.

- “Good” reusable artifacts can only be achieved after they have been evolved and adapted several times.
- It is inconceivable to predict all possible uses of a component upon its conception.
- Because one of the key characteristics of a reusable artifact is that it is reused in a (large) number of places, the likelihood of change propagation is much higher than to an ordinary component. Not only will there be other reusable artifacts that depend on it, but changes might be needed as well in all applications that have incorporated the reusable artifact!

Finally, all the issues mentioned above relate to technical problems with software evolution. Apart from that, there are also many organisational and managerial problems. These are however beyond the scope of this dissertation.

II 4.2 KINDS OF SOFTWARE EVOLUTION

In research literature, an important distinction is made between two kinds of software evolution. With *run-time* evolution (also called *autonomous* or *programmed* evolution), the software is dynamically modified while the application is running. With *design-time* (or *heteronomous*) evolution, on the other hand, changes are made manually by a software engineer during the software development process.

With *heteronomous* evolution, the changes to a software artifact can be totally unpredictable. As a result, it is an illusion to create fully automatic tools that perform these changes, and ensure that the resulting software artifact is consistent and conflict-free. Nevertheless, to aid in the process of evolution, sophisticated tools such as version management systems, reuse repositories, conflict detection mechanisms and refactoring tools can be applied. In practice, however, there will always remain some actions that need to be performed manually.

With *autonomous evolution*, software artifacts can change themselves *automatically* when receiving triggers which activate evolution. This kind of evolution poses many additional technical questions, and requires meta-computational mechanisms such as reflection, higher-order functions, partial evaluation and dynamic computation to allow software modules to change themselves. An intrinsic aspect of autonomous evolution is that it can only be used to deal with *anticipated changes*. As a result, the approaches can resort to more powerful conflict resolution techniques like automatic deadlock detection and consistency checking. In the area of architectural evolution, the autonomous approach is usually preferred [Kramer&Magee98, Wermelinger98].

II 4.3 EVOLUTION IN THE SOFTWARE LIFE-CYCLE

Like reuse, software evolution is not restricted to the implementation phase only. Even in the earlier phases of requirements specification, analysis and design, evolution is a strict necessity. Until now, most research on evolution has been dedicated to the implementation phase, and to a lesser degree in the earlier phases of requirements specification and design. However, there is a tendency to shift towards earlier phases. An enumeration of some of these approaches is given below:

- One way to add support for evolution at the requirements level, would be to make use of *change cases* [Ecklund&al96]. Change cases are descriptions of future requirements, and indicate potential directions of future development. They are expressed in use case notation [Jacobson&al92], and augment the application’s use case model.
- In [Wiels&Easterbrook98] a formal approach is taken to manage evolving specifications by making use of category theory. The proposed formalism allows to reason about the impacts of change on interconnected components, and also supports compositional (or incremental) verification.
- *Reuse cases* [Butler97] are a way to document object-oriented frameworks and how these frameworks are reused (or customised). Put simply, a reuse case is a use case that documents the different steps that need to be taken when customising a framework. However, since frameworks can be customised in different ways, different types of reuse cases are needed. They can be categorised as *composing*, *extending*, *flexing*, *evolving* and *mining* reuse cases.
- Though initially developed for dealing with reuse and evolution of implementation, *reuse contracts* have also been applied to evolution of software requirements [D’Hondt98], as well as evolution of UML collaboration diagrams [Mens&al99a].

- Last but not least, in the area of software architectures, residing in the architectural design phase, many attempts are being made to add support for *architectural evolution* [Kramer&Magee98, Wermelinger98]. Sometimes this is also referred to as *architectural reconfiguration*.

II 4.4 CO-EVOLUTION

When evolution takes place during different phases of the software development process, an additional problem arises. Whenever the implementation of the software evolves, the models in the earlier phases should be kept in sync by resorting to compliance checking techniques. Conversely, if one of the models in the earlier phases evolves, the implementation should be modified accordingly. This problem is sometimes called *co-evolution*, indicating that all software artifacts in the different phases of the software life-cycle should evolve simultaneously, and should be kept consistent with each other as much as possible.

In the idealised case, the implementation can be kept consistent with models in earlier phases by performing automatic code generation. In practice, however, this ideal case rarely occurs, and impact analysis or conflict detection techniques are required to identify those pieces of code that still need to be modified manually.

Several approaches towards co-evolution have been proposed. Below, we focus on two of them.

II 4.4.1 Lattice-Theoretic Approach Towards Co-evolution

In [Katayama98], a lattice-theoretic formulation of the problem of co-evolution is proposed: given a software specification *S*, a program *P* can be created that satisfies this specification, either heteronomously by means of a software development process (analysis, design and implementation) or autonomously by automatically deriving the program from the formal specification. When the specification evolves into a new version *S'*, the program should be modified accordingly. Similarly, when the program *P* evolves into a new version *P'*, the specification should be kept in sync.

Instead of allowing the specifications and programs to evolve in arbitrary ways, *evolution relations* $\subseteq_S$ and $\subseteq_P$ are introduced. These relations induce a mathematical lattice over the set of all specifications and the set of all programs, respectively. The key to a solid treatment of evolution lies with identifying relevant evolution relations. For example, the evolution relation can correspond to *functional augmentation*, as is the case for most variants of inheritance. As another example, the evolution relation can represent *refinement* or *reification*, which corresponds to instantiation in object-oriented application frameworks.

While a clear advantage of this approach is that it is not necessarily restricted to the object-oriented paradigm, an inherent shortcoming of the formalism is that the evolution relation always assumes incrementality of specifications, i.e., if $S \subseteq_S S'$ then *S'* extends the functionality or behaviour of *S*. Another restriction of the approach is that it only mentions specifications and programs, but not the intermediary phases of analysis and design.

II 4.4.2 Service Channels

In [Pirker&al98], *service channels* are proposed to keep the specification and implementation of object-oriented programs in sync. The specifications are given in Object-Z [Carrington&al90, Duke&al91]. Basically, a service channel is a mechanism for relating a sequence of transformations on the specification level to the implementation level.

In order to cope with both autonomous and heteronomous evolution, a distinction is made between *built-in* service channels and *external* service channels.

II 4.5 GRAPH-BASED APPROACHES TOWARDS EVOLUTION

This section presents an overview of the graph-based approaches towards evolution that can be found in the literature. The essential idea is that graphs can be used to describe relationships between software entities in an intuitive and visual way. Most existing graph-based approaches (such as dependency analysis, traceability analysis and impact analysis) can be (and have been) used to build tools and techniques to support the software evolution process.

II 4.5.1 Dependency Analysis

Software maintenance is a major problem in large pieces of software. In object-oriented programming, this problem manifests itself in code that is distributed over a large number of methods, that may reside in many different classes. In [Wilde&al89], the *dependency analysis* paradigm is proposed to express the information that a maintainer needs to know in order to understand the structure of a program. This paradigm looks at a software system as a collection of entities with dependencies between them. A distinction is made between data-flow dependencies, definition dependencies, calling dependencies, and functional dependencies. All these dependencies are stored in a *dependency graph*, in which each node represents a program entity and each edge represents a dependency between entities.

In object-oriented software systems, many more dependency relationships can be distinguished [Wilde&al92]. For example, there are class-to-class relationships such as *inheritance* and *uses*, class-to-method relationships such as *implements* and return types, class-to-message relationships, class-to-variable relationships, method-to-method relationships, etc…

An additional problem with object-oriented code is the fact that polymorphism allows many different methods with the same name and usually similar behaviour. However, in some cases, the behaviour of different methods with the same name can be significantly different, thus misleading the maintainer of the software. These inconsistent naming conventions can lead to very subtle bugs during reuse or evolution of these methods. To solve this problem, *external dependency graphs* are proposed in [Wilde&al92]. These graphs represent the effects of execution of a method using some form of data-flow. If methods with the same name have different external dependency graphs, this may indicate potential problems during reuse or evolution. As a concrete example, Wilde and Huitt illustrate the *at: put:* method in the Smalltalk class hierarchy, which is implemented in 14 different classes, with 4 different external dependency graphs. This analysis allows to pinpoint places where potential conflicts might occur.

While the dependency analysis paradigm mainly uses dependency graphs to capture and deal with data dependencies, there are also *program-dependence graphs* that represent control dependencies [Ottenstein84]. These dependencies represent relationships among program statements that control program execution. They are usually needed to perform *control flow analysis*.

II 4.5.2 Traceability Analysis

The approaches mentioned above use graphs to represent data and control dependencies and perform dependency analysis at the implementation level. However, the ideas mentioned there are much more general, in the sense that they can also be applied to higher levels, such as analysis and design. An even further generalisation is to use graphs to represent dependencies between artifacts residing at different levels, for example to relate a requirements specification to an associated design component. The latter approach is usually referred to as *traceability analysis*.

In [Bohner96], a distinction is made between *horizontal traceability* and *vertical traceability*. Horizontal traceability addresses dependency relationships between software artifacts in different phases of the software life-cycle, while vertical traceability restricts itself to expressing dependencies between components within the same phase. In this sense, horizontal traceability is the same as dependency analysis, but not necessarily restricted to the implementation phase.

One possible application of traceability is *program understanding*, since dependencies give us more information about the relationships between software artifacts. Without explicit traceability, many of these relationships would remain implicit, and it would be much harder and more time-consuming to understand a given piece of software. It is needless to say that traceability information can easily be integrated in browsing tools. Another important application of traceability is *impact analysis*.

II 4.5.3 Impact Analysis

It is obvious that a graph representation of dependencies between arbitrary software artifacts (within the same phase or between different phases) can be used immediately to perform *impact analysis*. When a particular artifact changes, all other artifacts that depend on it, either directly or indirectly, might require changes as well. [Bohner&Arnold96b] describe how dependency graphs can be used for impact analysis:

> *The graph nodes represent the information in software work products. Each work product contains a node for each component. The arcs represent dependency relationships both among components within a work product and among the work products themselves. The start*

> *node is the changed component (such as a changed requirement), the end node is the component affected (such as a design object), and the arc between them indicates that the end node depends on the start node. The nature of the relationship is captured as a label on the arc.*
>
> *In a graph representation of dependencies, a direct impact occurs when the affected component is related by a dependency that directly connects a related component. This type of impact, also called a first-order impact, can be obtained from the connectivity graph. An indirect impact occurs when the object affected is related by the set of dependencies representing a path between the component and the affected object. This type of impact is also referred to as an N-order impact, where N is the number of intermediate relationships between the component and the affected object.*

Software change impact analysis estimates what will be affected in software and related documentation if a proposed software change is made. Impact-analysis information can be used for planning changes, making changes, accommodating particular types of software changes, and tracing through the effects of changes. Impact analysis provides visibility into the potential effects of changes *before* the changes are implemented. This can make it easier to perform the changes more accurately. A state of the art overview of research literature on impact analysis can be found in [Bohner&Arnold96a].

A part of the impact analysis research that is worthwhile noting is devoted to *effort estimation*, which tries to estimate in advance the effort (or time) required to make a particular software change. This problem is far from trivial, since seemingly minor software changes are often much more extensive (and therefore more expensive to implement) than expected.

II 4.5.4 Adaptive Programming

In [Lieberherr&al93], *adaptive programming* is introduced as an extension to conventional object-oriented programming. Adaptive programming facilitates expressing the elements that are essential to an application by avoiding to make a commitment on the specific class structure of the application. Only the essential details – classes and methods – of the program need to be specified. Essential in the approach is the use of *traversal strategy graphs* [Lieberherr&Patt-Shamir97], called *strategies* for short. These strategies add a new layer of abstraction to object diagrams and class diagrams, in the sense that for each specific purpose only the essential dependencies in these diagrams are dealt with, while the rest is filtered out. For example, when dealing with class collaborations, one does not want to refer to all the details of the class graph but only want to identify minimal properties that the class graph must have, such as required existence of particular paths.

II 4.5.5 Version Graphs

In section II 4.1 we already mentioned the need for a version management mechanism to deal with certain problems related to software evolution. Many approaches make use of *version graphs* to represent the version space. These version graphs may be sequential, tree-structured, or acyclic. Clearly, the latter one is more general. It can be found in tools like PCTE [Oquendo&al89]. An even more general approach is a *two-level organization*, where the version graph is composed of several branches, each of which consists of a sequence of revisions. Revisions in each branch can be related to revisions in other branches using different kinds of relations such as offspring, successor and merge. This is the approach taken in ClearCase [Leblang94].

II . 5 REUSE CONTRACTS

> *"Formalising contracts, where possible and agreeable, is a good idea. However, attempting to formalise everything can easily lead to totally unapproachable and therefore unsaleable situations. It is important that a construct does not overspecify a situation. As in the real world, the enforcement of unnecessary requirements causes costs to increase dramatically and feasibility to diminish. The art of keeping contracts as simple as possible, but no simpler, has yet to develop in the young field of software components." [Szyperski98]*

This section takes a closer look at reuse contracts, since this is the approach towards evolution that will be used in this dissertation. Only an intuitive explanation of the issues and terminology of reuse contracts is given here, as well as an illustrative example. The exact formal definition of reuse contracts will be postponed to a later chapter.

II 5.1 WHAT IS A CONTRACT?

Throughout the years, the term **contract** has been used in many different ways in the object-oriented research community. All these different uses have in common that a contract is essentially considered to be a collection of participants, together with a number of obligations that need to be fulfilled by these participants. Next to these obligations, the contract may also express permissions, prohibitions and guidelines.

Because of the plethora of different meanings for the term contract, below we give an overview of the ones most commonly used, and show how the various interpretations differ from one another.

A *class interface* specifies the pre- and postconditions for a particular class, possible class invariants, (type) constraints given by the signature of a method, and the interface semantics of the method. In [Meyer92], this term is referred to as an **interface contract**. Indeed, an interface specification can be considered as a contract between the client (user) of an interface and the provider (or implementer) of the interface specification. Different kinds of interface contracts may be needed for different kinds of persons. The *client interface* is needed for clients that want to interact with the given class by sending messages. The *specialisation interface* [Lamping93] is needed for inheritors (or specialisers) that want to create a subclass of the class.

In [Larman98] and other work, **operation contracts** are used to describe the declarative behaviour of an operation. An operation contract describes what an operation commits to achieve. Since it is declarative in style, it emphasises *what* will happen, rather than *how* it will be achieved. Usually, operation specifications are expressed in terms of pre- and postconditions state changes. In this perspective, the client (or caller) of the operation has to make sure that the preconditions are satisfied, and can rely on the fact that the postconditions will be valid afterwards. Conversely, the provider (or implementer) of the operation has to make sure that the postconditions become true before returning to the client. To achieve this, the implementer can rely on the preconditions of the operation.

An **interaction contract** [Helm&al90] describes the interaction between a set of collaborating classes. In this sense, it is a contract between the different participants in the collaboration. This contract describes how the different participants interact by means of behavioural dependencies. Moreover, each participant in the contract needs to specify its class interface. Interaction contracts also contain preconditions required to establish the contract, and invariants to be maintained by these participants. At a different level, an interaction contract can be seen as a contract between the user of a collaborating class component and the implementer of this component.

Similarly, in [DeHondt98], **collaboration contracts** are introduced to describe the collaboration between classes. It can be considered as a simplification of the interaction contracts above, in the sense that preconditions, postconditions and invariants are not specified. The only essential information is the associations and message sending behaviour between the different classes.

A **reuse contract**, as originally defined in [Steyaert&al96], and later refined by [Lucas97], is essentially a contract between the provider and a reuser of an evolving component. The actual form of an evolving component can vary depending on the domain in which reuse contracts are used. The component that evolves, may be a single class [Steyaert&al96], a set of collaborating classes [Lucas97], a UML interaction diagram [Mens&al99a] or any other evolvable component. The provider has the obligation to describe how the component can be reused, while the reuser has the obligation to describe how the component is actually reused. This is expressed by means of a *contract type* [Mens&al98a] that specifies the kind of modification that takes place.

In [Mezini97] the term **co-operation contract** is coined as a synonym for reuse contract. In the context of class inheritance, a co-operation contract specifies properties of the base class to be propagated to inheritors. Some of this properties are strict, in the sense that they must be satisfied by inheritors, while other properties are negotiable in the sense that they may be accepted or rejected by the inheritor. After negotiation (which can, among others, be performed interactively), a meta-level application ensures that the inheritance mechanism is modified to cope with the new base class. This is referred to as *smart composition*.

In the remainder of this section, we discuss the idea of reuse contracts in more detail, and we start by motivating the need for reuse contracts.

II 5.2 Fragile Base Class Problem

Historically, the need for reuse contracts arose in situations such as the so-called *fragile base class problem*, where independently developed subclasses of a given base class can be broken whenever the base class evolves.

In the literature, the fragile base class problem is interpreted in two different ways. The term *syntactic fragile base class problem* is used when only the interface of the base class is modified, while a *semantic fragile base class problem* occurs when the implementation of the base class is changed as well. Actually, this distinction is closely related to the difference between interface inheritance and implementation inheritance, as for example present in Java.

In the SOM approach developed by IBM [IBM94], the syntactic variant is dealt with, allowing (in some cases) a base class interface to be modified without needing to recompile clients and derived classes dependent on that class. This is for example the case when performing a behaviour-preserving refactoring.

The question remains, however, how a subclass can remain valid in the presence of different versions and evolution of the *implementation* of its superclasses? Even worse, both variants of the fragile base class problem often occur together, when both the interface and the implementation change. Therefore, separating the syntactic from the semantic problem is questionable.

The reuse contract formalism tries to deal with both variants of the fragile base class problem simultaneously, in a uniform way. Moreover, although reuse contracts were first developed in [Steyaert&al96] to deal with evolution of classes only, the ideas are much more general, and can be applied to any situation where a component evolves while other components depend on it.

II 5.3 Terminology

Although the term *reuse contracts* is used, its underlying ideas are not restricted to *reuse* only, but are also applicable to express *evolution* of software components. Actually, the term “evolution contract”, or “modification contract”, or “adaptation contract” would have been more appropriate. Nevertheless, the term “reuse contract” is kept for historic reasons, since this was the term that was introduced in [Steyaert&al96], the first paper about reuse contracts.

The essential idea behind reuse contracts is that component reuse and evolution is based on an explicit *contract* between the *provider* of a component and a *modifier* (i.e., a *reuser* or an *evolver*) that incrementally modifies this component. The purpose of this reuse contract is to make reuse and evolution more disciplined. To achieve this goal, both the provider and the modifier have *contractual obligations*. The primary obligation of the provider is to document on which of its properties modifiers can rely. This is specified in a so-called *provider clause*. The *modifier clause* on the other hand documents how the component is actually being modified (during reuse or evolution).

To be able to express the specific way in which a component is modified, different kinds of modification need to be identified. Each of these different ways is specified by a *contract type*. Possible contract types are extension, cancellation, refinement and coarsening. These types impose obligations, permissions and prohibitions onto the modifier. Contract types are fundamental to disciplined reuse and evolution, as they form the basis for detecting conflicts when provided components are modified. Note that the terminology of *contract type* differs from [Lucas97], where the term *reuse operator* was used.

II 5.4 Goals and Design Considerations

Reuse contracts help in managing component evolution. Reusers can benefit from improvements to the components they reuse, and the proliferation of different versions of reusable components can be kept to a minimum. The contract types allow a developer to assess the impacts of change, and to decide whether changes should be made. Moreover, contract types provide developers with a vocabulary to discuss reuse and evolution, and assist them in better understanding the structure and behaviour of the systems they work with. As stated in [Lucas97], the main contributions of the reuse contract approach are:

- Reuse contracts can be used as structured documentation of reusable components, and generally assist a software engineer in adapting components to particular needs.
- Reuse contracts encourage disciplined reuse and evolution without being too coercive. Moreover, they provide a vocabulary and notation to discuss reuse and evolution.
- During evolution, reuse contracts assist in assessing how much work is necessary to update previously built applications, where and how to test, and how to adjust the applications. More specifically, reuse contracts allow us to detect incompatibilities when a certain component is upgraded to a new version, while other components still depend on it.

Besides the above advantages, reuse contracts can also be very beneficial during collaborative software development, when a large team of software developers maintains and modifies the same software. In this situation, parallel modifications to the same part of the software frequently occur, and mechanisms are needed to detect potential inconsistencies when these parallel evolutions are merged in a new version of the software.

Several design considerations have been taken into account when developing the reuse contracts methodology:

- Reuse contracts try to find the *balance between ease of use and formality*. Formality is needed, among others to facilitate automatic tool support. On the other hand, the model should not be too formal, in order not to discourage software engineers in using the methodology. Indeed, despite their many virtues, the most important reason why formal specification languages have never seen a major breakthrough, is probably because they were too formal.
- Another design consideration is that reuse contracts need to be able to deal with *unanticipated changes*. When only foreseen changes can be made, the way in which a software component can evolve is severely restricted. On the other hand, because of this, more assumptions can be made to facilitate the detection and resolution of evolution conflicts. Since this is not the case with reuse contracts, we will not always be able to detect all evolution conflicts, and we cannot provide fully automated support for solving these conflicts.
- A third design consideration is that reuse contracts focus on *the essential aspects of software only*. Depending on the domain to which reuse contracts are applied, this can mean different things. In [Steyaert&al96] reuse contracts were employed to deal with evolution of classes, and so-called *specialisation interfaces* [Lamping93] were used to specify the calling structure between the different methods in a class. However, instead of documenting all possible method calls, only the ones that were important from some point of view were mentioned. In [Lucas97] reuse contracts were applied to class collaborations, and only the essential message interactions between particular classes were mentioned. Moreover, classes could play a different role in different collaborations, allowing us to focus on different aspects of the class without needing to look at the entire class at once. To summarise, focussing on the essential aspects only is important, since it allows one to look at software at a higher level of abstraction, and consequently makes the software more reusable.

II 5.5 Historic Overview

Since the research in this dissertation is entirely devoted to reuse contracts, it is necessary to clarify how this dissertation fits in the reuse contract research. This is achieved by giving a short historic overview.

In [Steyaert&al96], reuse contracts were introduced for the first time, as a means for dealing with the fragile base class problem in object-oriented class hierarchies. In the Ph. D. dissertation of Carine Lucas this research was extended and generalised to deal with evolution of collaborating classes [Lucas97], and the general terminology behind reuse contracts was established. [Mezini97] applied meta-level programming to deal with the fragile base class problem, and provided an extension to reuse contracts in that some evolution conflicts could even be resolved in a semi-automatic way!

In order to test the generality of the ideas behind reuse contracts, [D'Hondt98] applied them to a completely new domain, namely software requirements in Object Behaviour Analysis [Rubin&Goldberg92], one of the many approaches to do requirements analysis. In [Mens&al98a] and [Mens&al99a] the ideas of reuse contracts were integrated in UML to deal with evolution of collaboration diagrams. A generalisation of this appeared in [Mens&al99b], where it was shown how to provide support for evolution of all kinds of UML diagrams by extending the UML metamodel directly.

From a more pragmatic side, [Codenie&al97] indicated that reuse contracts can provide help when evolving a real-world object-oriented application framework. Continuing in this direction, Koen De Hondt showed in his Ph. D. dissertation how reuse contracts can be integrated in a software development environment, and more importantly, how class collaborations and reuse contracts for these collaborations can be reverse engineered from source code [DeHondt98]. The key idea in this research was to view both collaborations and reuse contracts as instances of the much more general concept of software classifications.

In this dissertation, the reuse contract research is tackled from a more formal side. More specifically, it is investigated how most of the ideas mentioned above can be formally defined. One direct benefit of this formal approach is that it brings us one step closer to *automated support for software evolution*. Moreover, thanks to a formal foundation, we do not have to restrict ourselves to one specific instance of evolution, e.g., evolution of class collaborations. Instead, we can consider our formal foundation *as a general tailorable framework* that can be instantiated or customised in any domain of software engineering where support for reuse and evolution is desired. Another added value of a formalism is that it allows us to *simplify the model* significantly. For example, we can reduce the number of primitive contract types, and consequently also the number of reuse conflicts without loss of information. More important results such as a *normalisation algorithm* for sequences of evolution steps allow us to facilitate conflict detection. These important results immediately validate the practical relevance of this work.

Finally, note that the emphasis in this dissertation lies on *evolution* (as opposed to *reuse*, as was the case in [Lucas97]). More specifically, we will focus on support for collaborative development, where different software developers modify the same software in parallel.

II 5.6 An Example

In order to explain the ideas behind reuse contracts more clearly, we will work out a simple example, taken from [Mens98]. Although this example is expressed in UML notation, the ideas are simple enough to be meaningful to readers not familiar to UML as well.

Figure 2 expresses the essential design for navigation in a web browser. There are only two participating interfaces in the collaboration: `Browser` and `Document`. These communicate with each other through an association with two roles: `browser` and `doc`. `Document` contains two operations: `mouseClick` and `resolveLink`. `Browser` also contains two operations that are important for navigation: `handleClick` and `getURL`. When a mouse click is detected by the browser, the `handleClick` operation is invoked. This operation detects whether the click occurs inside a document. If this is the case, the browser sends a `mouseClick` message to `Document`, which determines if this mouse click causes a link to be followed. If this is the case, the `resolveLink` self send is issued. `resolveLink` specifies what happens when a hyperlink is followed in the document, and sends a message `getURL` back to `Browser` to fetch the contents of the web page pointed to by the hyperlink.

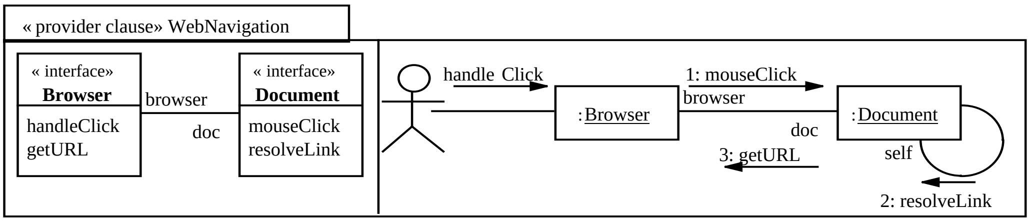


Figure 2: WebNavigation component

Figure 3 shows a user-defined customisation of the `WebNavigation` component, where `Document` is specialised to a new kind of document (a PDF document) that contains hyperlinks that point only to places within the document itself. For this reason, the targets of these links can be retrieved by the document itself. This is achieved by removing the `getURL` invocation and replacing it by a `gotoPage` self send. In object-oriented languages this kind of customisation (or specialisation) can easily be achieved by creating a subclass `PDFDocument` that inherits from the original `Document` class. This subclass adds a new `gotoPage` operation, and overrides the `resolveLink` operation.

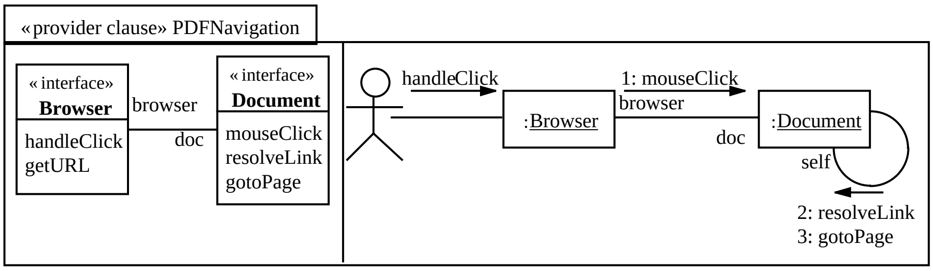


Figure 3: PDFNavigation component

Now suppose that the original `WebNavigation` provider clause evolves into `HistoryNavigation` by adding history behaviour (Figure 4). As a result of this, each time a hyperlink is followed through `getURL`, the URL of this link is stored somewhere through an extra invocation of `addURL`. This allows us to return to this location at a later time.

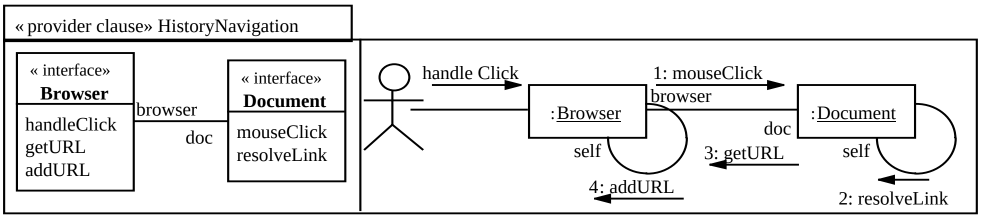


Figure 4: HistoryNavigation component

Both modifications work fine separately, but an evolution conflict arises when we try to combine the specialised PDF document behaviour with the evolved history functionality. Since link resolving is dealt with by the PDF document itself, `resolveLink` no longer invokes `getURL`. As a result, the `addURL` operation in `Browser` will never be invoked, so the history will not be updated when a link is followed within the `Document`. This conflict is called an *inconsistent operations* conflict, since the `resolveLink` operation becomes inconsistent with `getURL`.

In order to detect this conflict automatically, we need to document the changes that were made to the original `WebNavigation` provider clause, in the reuse step as well as in the evolution step. Schematically, all these changes are illustrated in Figure 5. It is composed from two different reuse contracts. The first reuse contract consists of the `WebNavigation` provider clause, while the horizontal arrow represents the modifier clause. The combination of both contract clauses determines how the `WebNavigation` component is modified into a new version called `HistoryNavigation`. In a similar way, the second reuse contract contains the same provider clause as the first while the vertical arrow represents the modifier clause. The combination of both clauses specifies how `PDFNavigation` is obtained as a customisation of `WebNavigation`.

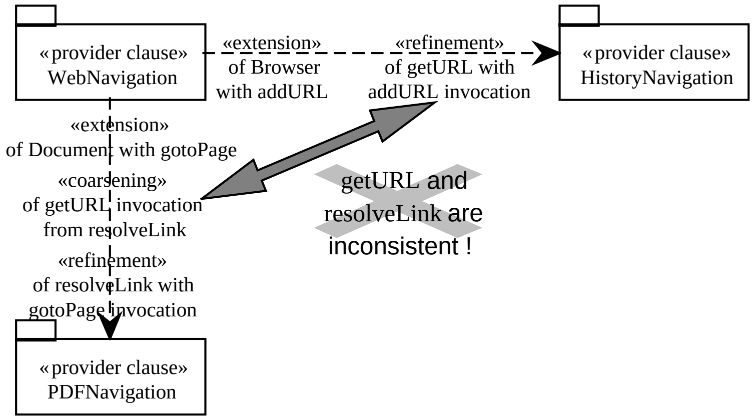


Figure 5: Documenting reuse and evolution with contract types

The modifier clauses of both reuse contracts in Figure 5 can be considered as sequences of elementary modification steps, and each of these elementary steps is specified by a contract type. The vertical modifier clause contains three contract types: an *extension* (to add an operation to a class), a *coarsening* (to remove an operation invocation) and a *refinement* (to add an operation invocation). For the horizontal modifier clause we have an *extension* and a *refinement*.

If two reuse contracts have the same provider clause, as is the case in Figure 5, evolution conflicts can be detected by a pairwise comparison of each of the primitive contract types in both modifier clauses. In this specific example, the horizontal *refinement* and the vertical *coarsening* lead to a conflict because they make incompatible changes that involve the same operation `getURL`. Chapter IV gives a detailed treatment of all kinds of evolution conflicts that can occur. Therefore, a formal definition of reuse contracts is needed. Before this can be done, however, some more theoretical background about graphs and graph rewriting is presented in chapter III, since this is the formalism on top of which reuse contracts will be defined.

II 5.7 Relation to Other Techniques and Approaches

By explicitly documenting reuse and evolution in a disciplined way, reuse contracts provide support for change propagation and impact analysis. Reuse contracts also try to detect potential conflicts during evolution. This is different from many theoretical approaches towards evolution that take a conservative approach by trying to avoid evolution conflicts by restricting the way in which the software can evolve.

II 5.7.1 Autonomous Evolution and Co-evolution

In section II 4.2, a distinction was made between *autonomous* (or run-time) and *heteronomous* (or design-time) evolution. The first kind of evolution can only deal with anticipated changes, but because of this one can resort to more formal techniques such as deadlock detection. Reuse contracts are situated in the camp of heteronomous evolution, because one of the design considerations is the ability to deal with unanticipated changes. The other side of the coin is that automatic support for evolution becomes impossible. For example, when considering the detection of evolution conflicts, only *potential* conflicts can be detected in some cases, since one cannot always be certain of the intentions behind particular modifications. Also, the resolution of conflicts cannot be fully automated, although semi-automatic tools can be devised that provide support for this conflict resolution to a large extent.

Until now, reuse contracts have also only focussed on evolution of software artifacts in the same phase of the software life-cycle. Co-evolution of software artifacts in different phases has not yet been considered in the context of reuse contracts. It is clearly related to issues such as horizontal traceability and compliance checking, and certainly needs further attention. However, a formal foundation for reuse contracts is more urgently needed, so we will not focus on reuse contracts for co-evolution here.

II 5.7.2 Techniques Based on Traceability

One of the novel ideas behind reuse contracts is that an explicitly link or *trace* is maintained between the original an evolved version of a software component. In this way, techniques such as *dependency* *analysis*, *traceability analysis* and *impact analysis* become immediately generalisable to deal with evolving components as well.

The **key difference** between reuse contracts and these other graph-based techniques is that reuse contracts explicitly document the link between the original component (the provider) and its evolved version by means of a so-called *contract type*. In the case of *reuse*, this documentation describes the assumptions a reuser makes about the provided component. In the case of *evolution*, this documentation describes the parts of the component that are modified during the evolution step. Because this documentation is expressed in a formal way, it enables us to detect which assumptions that a reuser makes about a component become broken when the component evolves.

II 5.7.3 Managing Evolving Specifications

A related work which is of particular interest is [Wiels&Easterbrook98], where a category theoretical approach is proposed to manage evolving specifications. It is related to reuse contracts since it also reasons about impacts of a change on interconnected components. Like reuse contracts, it provides explicit support for traceability, as well as support for tracing the impacts of change. Additionally, the approach supports compositional (or incremental) verification. Actual results based on case studies are not yet available.

An important difference with reuse contracts is that the latter provide explicit feedback about *the kind of evolution conflict that are introduced*, so that conflicts can be dealt with in an appropriate way.

II 5.7.4 Transformational Approaches

Reuse contracts have in common with transformational approaches that they describe the evolution of a software artifact as a sequence of primitive modification steps (or transformation steps). In reuse contract terminology, the *contract type* serves to specify the kind of transformation that is being made. Below we specify a number of interesting object-oriented transformational approaches.

In object-oriented databases, schema transformations are used to evolve an object-oriented database schema. [Banerjee&al87] proposed a taxonomy of useful schema evolutions for evolving object-oriented database schemas.

Because a database schema bears strong resemblance with an object-oriented class diagram, the idea of applying transformations has also been used in object-oriented software development. In [Tokuda&Batory98] a set of object-oriented transformations is implemented as automated refactorings, i.e., behaviour preserving program transformations. This work is an extension of the Ph. D. dissertation of Paul Bergstein where a set of object-preserving class transformations were defined for class diagrams [Bergstein94]. The latter work was, in its turn, an extension of the Ph. D. dissertation of William Opdyke [Opdyke92] which was probably the first work to apply the ideas of [Banerjee&al87] to behaviour-preserving transformations for object-oriented applications.

II 5.7.5 Adding Layers of Abstraction

The fact that reuse contracts focus on the essential aspects of software only allows one to look at the problem at a higher level of abstraction. Actually these ideas appear in many different articles about reuse and even about software engineering in general. Without even pretending to be complete, some of the more widely known techniques are the use of design patterns [Gamma&al94], templates, meta-level programming, frameworks, software architectures [Garlan&Shaw96], and separation of concerns. However, two approaches in particular are worth mentioning.

The idea of *role-based modelling* is proposed in [Reenskaug&al96]. With this approach the focus is put on the use of object interactions (or class collaborations) instead of single classes. More importantly, the different roles played by objects can be described by means of different collaborations. In this way, the complexity of a software system can be reduced by looking at the important aspects only, just like in the reuse contracts approach.

A second interesting idea, which is more or less orthogonal to the previous one, is known as *adaptive programming* [Lieberherr&al93]. In this approach, the essential idea is that object-oriented programming is made more simple and more robust to changes (two seemingly conflicting goals) by avoiding making a commitment of the specific class structure of the application. Only the essential details – classes and methods – of the program need to be specified. All the rest is filtered out and can be dealt with indirectly by making use of *traversal strategies* [Lieberherr&Patt-Shamir97].

II 5.7.6 Merge Tools

One of the crucial areas where reuse contracts provide support is to detect inconsistencies when merging parallel evolutions of the same software artifact. From this point of view, reuse contracts can be considered as a sophisticated kind of *merge tool*. Essentially two different kinds of merge tools can be distinguished:

- A *two-way merge tool* compares two alternative revisions of the same software artifact and merges them in a single resulting version. To this end, it interactively displays the detected differences to the user who has to select the appropriate alternative. Alternatively, it may also perform an automatic merge, based on some arbitrary decision of which alternative is more appropriate.
- To reduce the number of decisions that have to be made by the user, a *three-way merge tool* consults a common ancestor version if a difference is detected. If a change has been applied in only one revision, this change is incorporated automatically. Otherwise, a conflict is detected that can be resolved either manually or automatically.

Three-way merging is more powerful than two-way merging because more information is available. If an item is only available in one of the two compared revisions, some two-way merge algorithms assume that this item has been added, and by default include it in the final merge result. However, this is not necessarily the desired behaviour. When the item has been deleted in one of the two revisions, but not in the other, a more appropriate behaviour would be to delete the item in the final merge result as well. Obviously, deletions can only be detected by three-way merge tools. Another situation where three-way merging is superior occurs when an item is encountered with different values in both revisions. By comparing the values with the one in the common ancestor, the merge tool can decide to retain the value that differs from the one in the ancestor.

It is obvious that reuse contracts belong to the three-way merge tools. For example, in Figure 5 we have two different revisions `PDFNavigation` and `HistoryNavigation` of the same base version `WebNavigation`. The contract types are used to detect evolution conflicts (or merge conflicts) between these parallel revisions.

In general, to detect merge conflicts (i.e., contradictory changes), three-way merging attempts to combine two different modifications of the same base version. A conflict arises if the two modifications do not commute (e.g., in the case of contradictory changes to the name of an operation). Merge tools can be categorized based on the semantic level at which merging is performed, and consequently, based on the kind of conflicts they can detect.

- *Textual merging* is applied to text files. Almost all commercial software configuration management systems support textual merging [Rigg&al95]. Although we can expect only an arbitrary text file as the result of the merge (instead of a well-formed software artifact) and only physical conflicts can be detected, textual merging seems to yield good results in practice [Leblang94]. In particular, it works well when small local changes to large well-structured programs are combined and changes have been coordinated beforehand so that semantic conflicts are unlikely to occur.
- *Syntactic merging* exploits the context-free (or even context-sensitive) syntax of the versions to be merged. Therefore, it can guarantee a syntactically correct result and can perform more intelligent merge decisions. However, syntactic merging has been realised only in a few research prototypes [Buffenbarger95, Westfechtel91].
- *Semantic merging* takes the semantics of programs into account [Berzins94, Binkley&al95, Horwitz&al89]. Semantic merge tools perform sophisticated analysis in order to detect conflicts between changes. However, it is a hard problem to come up with a definition of semantic conflict that is neither too strong nor too weak (and is decidable). Furthermore, the merge algorithms developed so far are applicable only to simple programming languages. For these reasons, semantic merge tools not (yet?) made their way into practice.

Reuse contracts can be categorised in the latter category of semantic merging.

A final distinction can be made between state-based merging and operation-based merging: Most existing merge techniques are *state-based*, in the sense that they only use the initial state and final state of an evolution step. On the contrary, *operation-based* merging [Lippe&vanOosterom92] also makes explicit use of the transformations that were applied to obtain the final state from the initial one. This is often better than state-based merging, since it provides better conflict detection and allows for better support for conflict resolution as well. Again, reuse contracts fall in this latter category of operation-based merging. A more detailed discussion about this is postponed until chapter IV.

II 5.8 Why Reuse Contracts?

There are several reasons why the reuse contracts approach was chosen for dealing with software evolution in a disciplined way.

The most important reason for choosing reuse contracts is that their practical use has already been investigated in different domains: object-oriented implementations [Steyaert&al96, Cornelis97], object-oriented design [Lucas97, Mens&al99a] and even object-oriented analysis [D'Hondt98]. This will facilitate the aim of proposing a domain-independent formalism for software evolution. It suffices to find the similarities between the different domains, and express them in a domain-independent way.

Earlier experiments in Smalltalk by Koen De Hondt have also indicated the direct applicability of reuse contracts in practical tools. This is very important if the ideas of this work need to be applied in practice. We will also show in this dissertation that reuse contracts make it possible to improve commercially available merge tools significantly. To this aim, the approach of operation-based merging [Lippe&vanOosterom92], which has shown to be a better alternative than most other approaches, is chosen and enhanced.

A final and more pragmatic reason for choosing reuse contracts is that the original conceivers of the methodology, Carine Lucas and Patrick Steyaert, were directly available.

II . 6 SUMMARY

II 6.1.1 Summary

This chapter discussed how reuse and evolution fit into the object-oriented development life-cycle. More specifically, an evolutionary development life-cycle was proposed, stressing the importance of iterative and incremental development.

The benefits of reusing artifacts in all phases of the life-cycle were summarised, as well as the many technical problems involved in reuse. Two well-known approaches for writing reusable software were discussed in some more detail, namely software frameworks and component-based development.

Next, the same approach was taken for software evolution. After having explained the major technical problems involved in software evolution, a brief overview was given of existing approaches towards evolution, with an emphasis on those that employ graphs as an underlying formalism.

Finally, the technique of reuse contracts was explained, motivated and situated in the context of software reuse and software evolution. It was also compared with other approaches to evolution. Reuse contracts provide help with change propagation, impact analysis and conflict detection. More specifically, they aid in detecting and resolving upgrade conflicts and merge conflicts.

II 6.1.2 What’s Next?

Until now a unifying formal model for reuse contracts has been missing. As a result, each time the ideas of reuse contracts were applied to a different domain, everything needed to be redefined from scratch:

- What are the reusable/evolvable components?
- Which kinds of components are there?
- How can simple components be composed to more complex ones?
- How can a component be modified (upon reuse or evolution)?
- How can simple modifications be composed to more complex ones?
- What are the possible relationships between components?
- What are the possible conflicts in related components when some of these components evolve?

While the answers to these questions are often partly specific to the domain to which reuse contracts are applied, previous work on reuse contracts has indicated that there are many similarities between all the different domains. For example, the basic ways to modify any software artifact recur in each domain, albeit sometimes in a different context. Also, a lot of the associated evolution conflicts arise in many different situations.

By defining a formal framework for reuse contracts, support for evolution can be obtained for every domain to which the framework is customised. Of course, some domain-specific items will still need to be specified manually, but most of the results will be inherited automatically from the formal framework.

In chapter IV, the formal framework for reuse contracts will be defined, on top of the formalism of conditional graph rewriting that will be introduced in chapter III. Chapter V will deal with some scalability (composability) issues of the proposed framework, while chapter VI will illustrate its domain-independence.

III . Graphs and Graph Rewriting

This chapter presents the underlying formalisms that have been chosen to deal with evolution of software. Labelled typed graphs are used to express arbitrary software artifacts, while conditional graph rewriting is needed to evolve these artifacts. Both formalisms are defined in terms of category-theoretical concepts.

III . 1 INTRODUCTION

This chapter defines the foundation of labelled typed graphs and graph rewriting, on which the reuse contract formalism will be built.

Section III . 2 formally defines *labelled typed nested graphs* and presents their graphic notation. These graphs will be used to represent arbitrary software artifacts in an abstract way. Only the elements and their relationships are considered important.

Section III . 3 continues with the formalism by introducing *graph rewriting* as a basis for expressing evolution of software artifacts. After some general definitions of graph rewriting, the formalism is extended to so-called *conditional* graph rewriting.

The next chapter builds upon this foundation to augment it with the formalism of reuse contracts, in order to obtain a general framework for detecting evolution conflicts. In chapter V, some important scalability issues are addressed. A validation of the graph formalism enhanced with reuse contracts is then made in chapter VI, where the domain-independent framework will be customised to different domains, to show that the principles behind software reuse and evolution are domain-independent.

Important note to the reader.

> *This is the most technical part of the dissertation. Before continuing with this chapter, it is advisable to read appendix VIII . 1 on page 210 first, since it contains the mathematical notations that will be used in this dissertation, as well as some elementary definitions about functions and relations, and a very brief introduction to category theory.*
>
> *Readers that are only interested in the intuitive ideas behind the thesis, may decide to skip this chapter. In the subsequent chapters, an intuitive explanation of the ideas is given as much as possible, so one can probably understand the main ideas of the dissertation without needing to dive into all the technical details.*

III . 2 DEFINITION OF LABELLED TYPED GRAPHS

This section formally defines labelled typed graphs. This is done gradually, by starting with basic graphs in subsection III 2.1, and adding labels, constraints and types to each node and edge of a graph in subsection III 2.2. Subsection III 2.3 introduces a nesting mechanism to reduce the complexity of graphs. Subsection III 2.4 enhances the basic notion of types by introducing a type graph which allows to put constraints between edge types and node types. Finally, a partial order is attached to node types and edge types, to be able to automatically inherit constraints from their supertypes.

As mentioned in the first chapter of this dissertation, an important aim of the thesis was *“**to deal with software evolution in a domain-independent way**”*. By using labelled typed graphs to represent software artifacts, it becomes very easy to represent artifacts in completely different domains, such as specification, analysis, design and implementation. It suffices to define a domain-specific labelling set and constraint set, define a domain-specific type graph which expresses the specific constraints that hold in the considered domain, and define a partial order on the valid node types and edge types in this domain.

III 2.1 GRAPHS

III 2.1.1 Motivation

Graphs are commonly known, well understood, have a firm mathematical basis (graph theory), and encompass a huge number of concepts, methods and algorithms. This makes them very interesting from a formal as well as a practical point of view.

We will use graphs to represent arbitrarily complex software artifacts and their interrelationships. Nodes of a graph can represent entities like methods, classes, objects, attributes, packages, components or even entire systems. The edges can be used to represent all kinds of relationships between these entities. This is essential, since the most important aspect of understanding a software system is understanding the different kinds of relationships between the different parts of the system [DePauw&al93].

Graphs can be used to describe and understand object-oriented programs, since they provide a compact and expressive representation of program behaviour. One of the earliest proposals was [Cunningham&Beck86], where graphs were introduced to describe the message sending behaviour between objects. This resulted in a better understanding of the Smalltalk-image, and facilitated debugging of object-oriented code. In [Kleyn&Gingrich88] different kinds of graphs were used to describe the behaviour of large scale object-oriented systems. Besides method invocation graphs, also object invocation graphs, taxonomy (or inheritance) graphs and part-whole graphs were introduced. Each kind of graph presents a different perspective on system behaviour, and each perspective yields different information. In this way, the behaviour of objects can be understood more easily, thus facilitating code sharing and reusability. In [Ellis95] the notion of conceptual graphs was applied to object-oriented concepts. Many object-oriented metrics [Chidamber&Kemerer91] are also based on a graph representation of the object-oriented system. Also in [Pfleeger&Bohner90], graph-based metrics are used to evaluate the maintainability of a system whenever a change is proposed.

A final important reason why we decided to use graphs is because we want to detect evolution conflicts between independently evolving software artifacts. As explained in [Steyaert&al97] and [Lucas97], most of the interesting evolution conflicts arise when existing software dependencies are inadvertently removed, when implicit assumptions are made about particular dependencies, or when particular dependencies between components are implicitly assumed without being actually present.

III 2.1.2 Basic Definitions

We first present the basic definition of graphs as can be commonly found in graph literature.

A **graph** G is a tuple $(V,E,source,target)$ such that:
1. V is a finite set of nodes (or vertices), and E is a finite set of edges (or arcs) such that $V \cap E = \varnothing$.
3. $source: E \rightarrow V$ and $target: E \rightarrow V$ are functions assigning exactly one source and target node to each edge.

Definition 1: Graph

<u>Remarks</u>:

- A graph G is **directed**, because each edge has exactly one *source* node and *target* node. A graph is a **multigraph** if different edges can have exactly the same source and target nodes. If the labelled graph contains no edges (i.e., $E = \varnothing$), it is called a **discrete graph**.
- When working with more than one graph, the abbreviations V_G, E_G, $source_G$ and $target_G$ are used to refer to V, E, *source* and *target*, respectively.
- Instead of using graphs, we could also have chosen for **hypergraphs**, which are slightly more general, in that they contain **hyperedges** that are allowed to have more than one source and target node. From a formal point of view, only the following two changes to the definition are required:

  $source: E \rightarrow V^+: e \rightarrow (v_1,\ldots,v_n)$ is a function assigning *a finite number* of source nodes (at least one) to each hyperedge.
  $target: E \rightarrow V^+: e \rightarrow (v_1,\ldots,v_n)$ is a function assigning *a finite number* of target nodes (at least one) to each hyperedge.

  The connection between graphs and hypergraphs is clear: a graph is a hypergraph with $\forall e \in E: |source(e)| = 1 = |target(e)|$. In practice, however, the additional expressiveness of hypergraphs is rarely needed. Even in those situations were we need to deal with hyperedges having multiple source and target nodes, they can always be reduced to a set of "ordinary" edges, by introducing a new intermediary node. This is the reason why we stick to ordinary graphs in this dissertation.

The graphs as defined above form objects in a category where the morphisms are functions that preserve the source and target of all edges. The proof of this is given in, among others, [Meseguer&Montanari90].

Let G and H be graphs. A **graph homomorphism** $f: G \rightarrow H$ is a pair ($f_{node}: V_G \rightarrow V_H$, $f_{edge}: E_G \rightarrow E_H$) such that:

$f_{node} \circ source_G = source_H \circ f_{edge}$ (source nodes are preserved)

$f_{node} \circ target_G = target_H \circ f_{edge}$ (target nodes are preserved)

Graph is a category with graphs as objects and graph homomorphisms as morphisms.

Definition 2: Category of graphs

The functions f_{node} and f_{edge} introduced above are usually called node mappings and edge mappings. Using these functions, the general category-theoretical notion of isomorphism can be simplified by stating that f_{node} and f_{edge} should be bijective functions.

A **graph isomorphism** $f: G \rightarrow H$ is a graph homomorphism such that f_{node} and f_{edge} are bijective functions. In that case, we say that ***G* is isomorphic to *H*** (denoted by $\boldsymbol{G \cong H}$).

Definition 3: Isomorphism of graphs

Actually, a graph isomorphism is nothing more than a graph homomorphism which is both *injective* and *surjective*. Especially the injective graph homomorphisms will play a special role in the remainder of this dissertation, since they allow to simplify many proofs.

A graph homomorphism f is **injective** if f_{node} and f_{edge} are injective. f is **surjective** if f_{node} and f_{edge} are surjective. ***InjGraph*** (resp. ***SurjGraph***) is the subcategory of ***Graph*** with graphs as objects and injective (resp. surjective) graph homomorphisms as morphisms.

Definition 4: Injective and surjective graph morphisms

It is trivial to check that ***InjGraph*** $\subseteq$ ***Graph*** and ***SurjGraph*** $\subseteq$ ***Graph***. Injective graph morphisms have the important property that no two different nodes or edges are mapped onto the same node or edge. They are important in the sense that they can be used to describe subgraphs. A graph is a subgraph of an other one if it has the same structure, i.e., if one can find an injective mapping of the nodes and edges such that the structure is preserved.

Let G and H be two graphs. H is a **subgraph** of G (denoted by $H \subseteq G$) if $\exists$ injective graph morphism $m: H \rightarrow G$ (called *match* of H in G).

Definition 5: Subgraph

As for functions, a graph morphism $f: G \rightarrow H$ is **total** if it is defined on its entire domain G. Otherwise it is **partial**, i.e., it is only defined on a subgraph of G. Moreover, ***Graph***^{P} forms a category with graphs as objects and partial graph morphisms as morphisms (see [Löwe93]). In a similar way, ***InjGraph***^{P} and ***SurjGraph***^{P} can be defined.

A graph morphism f is **total** iff f_{node} and f_{edge} are total. Otherwise, f is called **partial**.

Definition 6: Partial and total graph morphisms

III 2.1.3 Fan-in and Fan-out

We will now give some definitions that deal with the number of incoming and outgoing edges related to a particular node in a graph. Similarly, we can define all nodes that directly depend on a particular node, or all nodes on which a particular node depends.

Let G be a graph, and $v \in V_G$.

$\mathbf{InEdge_G(v)} = \{\, e \in E_G \mid target(e)=v \,\}$ $\mathbf{InNode_G(v)} = \{\, source(e) \mid e \in InEdge_G(v) \,\}$

$\mathbf{OutEdge_G(v)} = \{\, e \in E_G \mid source(e)=v \,\}$ $\mathbf{OutNode_G(v)} = \{\, target(e) \mid e \in OutEdge_G(v) \,\}$

$\mathbf{degree_G(v)} = |InEdge_G(v)| + |OutEdge_G(v)|$ $\mathbf{Adjacent_G(v)} = InNode_G(v) \cup OutNode_G(v)$

$\mathbf{fan\text{-}in_G(v)} = |InNode_G(v)|$ $\mathbf{fan\text{-}out_G(v)} = |OutNode_G(v)|$

Definition 7: Incoming and outgoing nodes

In the above definition, the ***fan-in***, or **in-degree** of a node counts the number of nodes on which a particular node depends. Similarly, the ***fan-out***, or **out-degree** of a node counts the number of nodes that depend on it. When considering traceability, the goal is to keep the *fan-out* of a node small, since it indicates the number of nodes that depend on a given node, and that are therefore likely to change whenever the given node changes. A similar reasoning can be made for the *fan-in*.

$InNode_G$, $OutNode_G$ and $Adjacent_G$ can be considered as *relations* on $V_G \times V_G$. Instead of defining a function $InNode_G$: $V_G \rightarrow \mathcal{P}(V_G)$: $v \rightarrow InNode_G(v)$ that maps each node on a possible empty set of nodes, we can define a relation $InNode_G \subseteq V_G \times V_G$ such that $(v,w) \in InNode_G$ iff $w \in InNode_G(v)$. A similar reasoning can be made for $OutNode_G$ and $Adjacent_G$. These relations are often easier to deal with from a practical point of view.

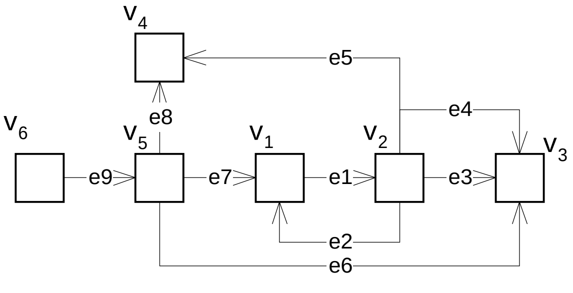


Figure 6: Graph example

As a running example throughout this chapter, consider the graph *G* of Figure 6. In this example,

$InEdge_G = \{ (v_1,e_2), (v_1,e_7), (v_2,e_1), (v_3,e_3), (v_3,e_4), (v_3,e_6), (v_4,e_5), (v_4,e_8), (v_5,e_9) \}$

$OutEdge_G = \{ (v_1,e_1), (v_2,e_2), (v_2,e_3), (v_2,e_4), (v_2,e_5), (v_5,e_6), (v_5,e_7), (v_5,e_8), (v_6,e_9) \}$

$InNode_G = \{ (v_1,v_2), (v_1,v_5), (v_2,v_1), (v_3,v_2), (v_3,v_5), (v_4,v_2), (v_4,v_5), (v_5,v_6) \}$

$OutNode_G = \{ (v_1,v_2), (v_2,v_1), (v_2,v_3), (v_2,v_4), (v_5,v_1), (v_5,v_3), (v_5,v_4), (v_6,v_5) \}$

When looking clearly at $InNode_G$ and $OutNode_G$ in this example, we see that $OutNode_G$ is the inverse relation of $InNode_G$, as formalised in the following property. Its proof immediately follows from the symmetry in the definition of $InNode_G(v)$ and $OutNode_G(v)$.

If *G* is a graph than $OutNode_G = InNode_G^{-1}$

Property 1: Symmetry of *InNode* and *OutNode*

III 2.1.4 Transitive Closure

Definition 68 of page 212 of the appendix can be used to define $InNode^2$ (or $OutNode^2$) for calculating all dependencies obtained by following a path of exactly 2 edges. In general, $InNode^+$ (or $OutNode^+$) can be used to calculate all indirect dependencies between nodes, by following a path of length one or more in the directed graph.

Let *G* be a graph, and $v, w \in V_G$.
G is **cyclic** if $InNode_G$ (or $OutNode_G$) is cyclic
w **(directly) depends on** *v* if $(v,w) \in InNode_G$
$\forall n \in \mathbb{N}_0$: *w* **n-depends on** *v* if $(v,w) \in (InNode_G)^n$
w **transitively depends on** *v* if $(v,w) \in (InNode_G)^+$. $(InNode_G)^+$ is called the **transitive dependency relationship**.
$\forall n \in \mathbb{N}_0$: ***fan-in****^{n}**(v)*** $= |InNode^n(v)|$ and ***fan-out****^{n}**(v)*** $= |OutNode^n(v)|$
fan-in*$^+$**(v)*** $= |InNode^+(v)|$ and ***fan-out****$^+$**(v)*** $= |OutNode^+(v)|$

Definition 8: Transitive dependencies between nodes

We say that *w* (directly) depends on *v* if there is an edge with *w* as source and *v* as target. In other words, the source of the edge corresponds to the *dependent* element. If *w* n-depends on *v*, then *v* can be reached from *w* by following a path of exactly n edges. We sometimes say that there is an **n-th order dependency** from *w* to *v*. *fan-in*^{n}*(v)* and *fan-out*^{n}*(v)* give a measure for the **n-th order impact** of *v*, namely the number of nodes that can possibly be affected by a change to *v*, because there is an n-th order dependency from these nodes to *v* [Bohner&Arnold96b].

The graph *G* in Figure 6 is cyclic, because $(v_1,v_1) \in InNode_G^+$ and $(v_2,v_2) \in InNode_G^+$. Indeed, $(v_1,v_2) \in InNode_G$ and $(v_2,v_1) \in InNode_G$, hence $(v_1,v_1) \in InNode_G^2$ and $(v_2,v_2) \in InNode_G^2$. Also, v_1 2-depends on v_3, since $(v_1,v_3) \in InNode_G^2$. In a similar way, v_6 2-depends on v_3. The transitive dependency on v_3 is $(InNode_G)^+(v_3) = \{v_1,v_2,v_5,v_6\}$. Consequently, $fan\text{-}in^+(v_3)=4$. Making a change to v_3 can possibly affect four other nodes.

Similar to the definition of transitive dependency on a node, which corresponds to the set of all nodes that depend on *v*, one can also define the *n-th order closure* and *transitive closure* of a graph. For this purpose we first need to give the definition of a *graph induced by a relation* that is defined on a set of nodes. For each relationship between two nodes, an edge will be generated in the induced graph.

Let *V* be a set of nodes and $R \subseteq V \times V$ a relation on *V*. $G = (V,E,source,target)$ is a **graph induced by *R*** if $\exists$ bijective function $f: R \rightarrow E: (v,w) \rightarrow e$ such that $\forall\ (v,w) \in R$: $source(f(v,w)) = v$ and $target(f(v,w)) = w$

Definition 9: Graph induced by a relation

It follows from the definition of relation that an induced graph is never a multigraph.

Let *G* be a graph. $\forall n \in \mathbb{N}_0$: the **n-th order closure *G*^{n}** is the graph induced by $(OutNode_G)^n$. The **transitive closure *G*$^+$** is the graph induced by $(OutNode_G)^+$

Definition 10: n-th order and transitive closure of a graph

Again this definition is a recursive one. Starting from all direct edges in *G*, the transitive closure calculates all indirect edges (or only up to a particular order n), by sequentially composing a number of edges. Note that, if the same transitive edge can be obtained in different ways, it will only be mentioned once in the transitive closure graph. Using a straightforward algorithm, calculating the transitive closure of a graph requires $\boldsymbol{O(n^3)}$ time, where *n* is the number of nodes in the graph [Warshall62]. In order to increase the efficiency, more sophisticated algorithms have been developed since. Unfortunately, they are only useful if the matrix representation of the transitive closure graph is sparse.

III 2.2 LABELLED TYPED GRAPHS

This section extends the basic notion of graphs with labels, constraints and a typing mechanism. *Constraints* on nodes or edges in a graph are used as a very flexible mechanism to express anything that cannot be expressed in any other way. The complexity of a graph is reduced by means of a *typing* mechanism. We can classify nodes and edges by attaching different types to them. Nodes and edges of the same type have the same characteristics. This is similar to the object-oriented approach, where all objects that have the same characteristics can be classified as a class that specifies these characteristics. Do not confuse the notion of types used here with the one that is used in programming languages.

III 2.2.1 Labelled Graphs

As a first extension to the previous definition of graphs, we attach a label and constraint set to each node and edge. Some examples of typical constraints will be given later.

Let $L = (NodeLabel, EdgeLabel)$ be a pair of disjoint, and possibly infinite, sets of labels. Let $C = (NodeConstraint, EdgeConstraint)$ be a pair of possibly infinite sets of constraints.
An ***(L,C)*-labelled graph** *G* is a tuple *(g,label,constraint)* such that:
(1) $g = (V,E,source,target)$ is a graph.
(2) $label = (vlabel: V \rightarrow NodeLabel,\ elabel: E \rightarrow EdgeLabel)$ is a pair of node-labelling and edge-labelling functions such that *vlabel* is an *injective* function.
(3) $constraint = (vconstraint: V \rightarrow \mathcal{P}(NodeConstraint),\ econstraint: E \rightarrow \mathcal{P}(EdgeConstraint))$ is a pair of node-constraint and edge-constraint mappings.
(4) $edge: E \rightarrow EdgeLabel \times V \times V: e \rightarrow (elabel(e),source(e),target(e))$ should be an *injective* function.

Definition 11: Labelled graph

<u>Remarks</u>:

- The definition of labelled graphs given above is the same as the one in [Ehrig&al91], except that our definition allows to attach a set of constraints to each node and edge. Moreover, we require some additional injectivity conditions on the node and edge labels. Although this is not necessary in general, for our specific purposes we want all nodes in a graph to have a unique label. The reason for this is that we will use the node labels as node *identifiers* (as opposed to many other approaches where the label represents the node *type*). Similarly, edges with the same source and target nodes should not have the same label, because they will be used as edge identifiers. Stated otherwise, an edge is uniquely determined by its label and the label of its source and target node. Hence the requirement that *edge* should be an injective function. An alternative definition of the edges would be $E \subseteq V \times EdgeLabel \times V$.
- For practical purposes, one can attach an **empty label** ε to edges.
- If the context is clear *label* is written instead of *vlabel* or *elabel*, and *constraint* instead of *vconstraint* and *econstraint*. This cannot cause any name conflicts since $V \cap E = \varnothing$. When dealing with more than one labelled graph at the same time, the abbreviations $label_G$, $constraint_G$ and $edge_G$ are used to refer to *label*, *constraint* and *edge*, respectively.

Similar to the categories ***Graph*** and $\boldsymbol{Graph^P}$, one can show that for any pair *L* (of labelling sets) and *C* (of constraint sets), the *(L,C)*-labelled graphs form a category as well. For this, the structure-preserving morphisms between labelled graphs need to be specified. Note that we make use of *partial* morphisms because they are more general. As a result, we will need to restrict ourselves to $dom(f_{node})$ and $dom(f_{edge})$ in the definition below.

Let G and H be (L,C)-labelled graphs.
$f: G \rightarrow H$ is a **labelled graph morphism** if $(f_{node}: V_G \rightarrow V_H, f_{edge}: E_G \rightarrow E_H)$ is a partial graph morphism between G and H.
An ***L*-preserving labelled graph morphism** $f: G \rightarrow H$ is a labelled graph morphism such that

$\forall v \in dom(f_{node})$: $vlabel_G = vlabel_H \circ f_{node}$ (node labels are preserved)

$\forall e \in dom(f_{edge})$: $elabel_G = elabel_H \circ f_{edge}$ (edge labels are preserved)

A ***C*-preserving labelled graph morphism** $f: G \rightarrow H$ is a labelled graph morphism such that

$\forall v \in dom(f_{node})$: $vconstraint_G = vconstraint_H \circ f_{node}$ (node constraints are preserved)

$\forall e \in dom(f_{edge})$: $econstraint_G = econstraint_H \circ f_{edge}$ (edge constraints are preserved)

An ***(L,C)*-preserving labelled graph morphism** $f: G \rightarrow H$ is an L-preserving and C-preserving labelled graph morphism.

Definition 12: Structure-preserving morphisms between labelled graphs

Given these four different graph morphisms, one can define four different categories in which the objects are (L,C)-labelled graphs. In the naming of each of these categories we use the convention to use a *prefix* to specify the objects we are dealing with (L if we work with labelled graphs), while a *postfix* is used for specifying the morphisms, i.e., the kind of structure that is preserved (LC if labels as well as constraints are preserved). Moreover, each of the categories is parameterised with the set L of labels and C of constraints.

LGraph(L,C) is a category with (L,C)-labelled graphs as objects and labelled graph morphisms as morphisms. ***LGraphL(L,C)*** is a category with (L,C)-labelled graphs as objects and L-preserving labelled graph morphisms as morphisms. ***LGraphC(L,C)*** is a category with (L,C)-labelled graphs as objects and C-preserving labelled graph morphisms as morphisms. ***LGraphLC(L,C)*** is a category with (L,C)-labelled graphs as objects and (L,C)-preserving labelled graph morphisms as morphisms.
Moreover, ***LGraphLC(L,C)*** $\subseteq$ ***LGraphL(L,C)*** $\subseteq$ ***LGraph(L,C)***,
and ***LGraphLC(L,C)*** $\subseteq$ ***LGraphC(L,C)*** $\subseteq$ ***LGraph(L,C)***.

Definition 13: Categories of labelled graphs

When the context is clear, the parameters (L,C) are usually omitted in the names of the categories. It is trivial to check the conditions which ensure that ***LGraph***, ***LGraphL*** and ***LGraphC*** and ***LGraphLC*** form categories, and that they are subcategories of each other.

Because of the injectivity constraints imposed on the labelled graphs in Definition 11, it turns out that label-preserving graph morphisms are always injective. This is an important property, since it will simplify many proofs. Another way to express this property using category theory, is by saying that there exists a forgetful functor F: ***LGraphL*** → ***InjGraph^P***.

If $f: G \rightarrow H$ is an ***LGraphL***-morphism, then f is injective.

Property 2: Injectivity of label-preserving graph morphisms

Proof:

$f_{node}: V_G \rightarrow V_H$ is injective because $\forall v, w \in dom(f_{node})$: $v \neq w$ implies $vlabel_G(v) \neq vlabel_G(w)$ (since $vlabel$ is injective). This implies $vlabel_H(f_{node}(v)) \neq vlabel_H(f_{node}(w))$ by definition of ***LGraphL***-morphism. As a result, $f_{node}(v) \neq f_{node}(w)$, since they have different labels.
$f_{edge}: E_G \rightarrow E_H$ is injective because $\forall e, f \in dom(f_{edge})$: $e \neq f$ implies $edge_G(e) \neq edge_H(e)$ which implies $elabel_G(e) \neq elabel_G(f)$ or $source_G(e) \neq source_G(f)$ or $target_G(e) \neq target_G(f)$. Each of the terms in the conjunction lead us to $f_{edge}(e) \neq f_{edge}(f)$ by using the definition of ***LGraphL***-morphism.

III 2.2.2 Labelled Typed Graphs

Besides labelling all the nodes and edges of a graph, an additional type will be attached to each node and edge to capture the similarities between particular nodes or edges. Note that, while edges with the same source and target nodes were not allowed to have the same label, they can have the same type.

Let *T = (NodeType, EdgeType)* be a pair of disjoint and finite sets of predefined types.
An ***(L,C)*-labelled *T*-typed graph** *G* is a pair *(g, type)* such that *g* is an *(L,C)*-labelled graph and *type = (vtype: V→NodeType, etype: E→EdgeType)* is a pair of functions attaching a type to each node and edge of the graph.

Definition 14: Labelled typed graph

Notation. If the context is clear we simply write *type* (instead of *vtype* and *etype*). When working with more than one graph, the notation $type_G$ is used instead of *type*.

As an example, reconsider Figure 6, but now with labels and types attached to the nodes and edges. For reasons of simplicity, we do not specify any constraints. Graph *G* in Figure 7 contains six nodes v_1, v_2, v_3, v_4, v_5 and v_6 with labels *a*, *b*, *c*, *d*, *e* and *f* respectively. Nodes v_1, v_2 and v_3 have type «υ», and nodes v_4, v_5 and v_6 have type «ω». Edge e_1 with label *p* and type τ connects node v_1 with v_2. Using the definitions above, $edge(e_1) = (p,v_1,v_2)$ and $type(e_1)=\tau$. Similarly, $label(v_1)=a$ and $type(v_1)=\upsilon$.

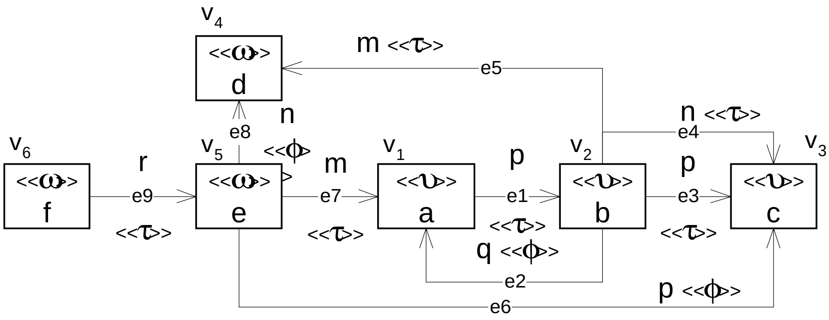


Figure 7: A labelled typed graph example

Let *G* and *H* be *(L,C)*-labelled *T*-typed graphs.
f: G→H is a **labelled typed graph morphism** if it is a labelled graph morphism between the *(L,C)*-labelled graphs *G* and *H*.
A ***T*-preserving labelled typed graph morphism** *f: G→H* is a labelled typed graph morphism such that

$\forall v \in dom(f_{node})$: $vtype_G = vtype_H \circ f_{node}$ (node types are preserved)

$\forall e \in dom(f_{edge})$: $etype_G = etype_H \circ f_{edge}$ (edge types are preserved)

Definition 15: Type-preserving morphisms between labelled typed graphs

LTGraph(L,C,T) is a category with *(L,C)*-labelled *T*-typed graphs as objects and labelled typed graph morphisms as morphisms. ***LTGraphL(L,C,T)*** contains *L*-preserving labelled typed graph morphisms as morphisms. ***LTGraphC(L,C,T)*** contains *C*-preserving labelled typed graph morphisms as morphisms. ***LTGraphT(L,C,T)*** contains *T*-preserving labelled typed graph morphisms as morphisms. In a similar way we define ***LTGraphLC(L,C,T)***, ***LTGraphLT(L,C,T)***, ***LTGraphCT(L,C,T)*** and ***LTGraphLCT(L,C,T)***.
Moreover, these categories are subcategories of each other in an obvious way.

Definition 16: Categories of labelled typed graphs

In the rest of this dissertation, the categories ***LTGraphL*** and ***LTGraphLT*** are used most often, depending on whether both the types and labels need to be preserved, or only the labels.

Nodes of a labelled typed graph are represented visually by using a rectangle surrounding the node information, i.e., label, constraints and type. The name of the node itself is mentioned outside the rectangle. Edges between nodes of a graph are depicted graphically by a plain line. The name of the edge is mentioned on this line. Because edges are directed, the target node of each edge should be pointed to by means of an arrow. Again, the label, constraints and type of each edge are mentioned by adding them above or below the edge line. The type of a node or edge is depicted between guillemets

«…». If present, constraints on a node or edge are specified between curly braces *{…}*. If the label of an edge is ε (i.e., the empty label), it is omitted in the graphical notation. In the graph L on the left-hand side of Figure 8, an edge e is visualised with $source(e)=1$, $target(e)=2$, e$type(e)=«\tau»$, e$label(e)=\varepsilon$, v$label(1)=A$, v$type(1)=«\upsilon»$, v$label(2)=B$ and $type(2)=«\omega»$.

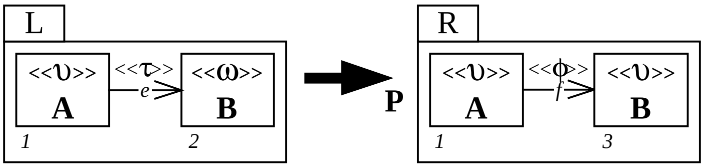


Figure 8: *LTGraphLT*-morphism

Because of the use of partial graph morphisms, there exist some ***LTGraphLT***-morphisms which do not appear to be type-preserving at first sight, although they are. For example, the ***LTGraphLT***-morphism $P: L \rightarrow R$ of Figure 8 does not appear to be type-preserving, since the morphism changes the type of node 2 from ω to υ, while it also changes the type of the edge e from τ to ϕ. This is not a problem, as long as the (partial) node- and edge-mapping functions P_{node} and P_{edge} are chosen as follows:

P_{node}: $\{1,2\} \rightarrow \{1,3\}$ is a partial injective function with $dom(P_{node})=\{1\}$ and $P_{node}(1)=1$

P_{edge}: $\{e\} \rightarrow \{f\}$ is a partial injective function with $dom(P_{edge})=\emptyset$

In practice, such a situation can occur when the node with label B is removed from L, while at the same time, a different node with label B (but a different type) is introduced in R. As a result, these are two different nodes, although they accidentally have the same label. Note that, in order to avoid dangling edges, which would breach the well-formedness of the graph, the edge in L needs to be removed as well (since its target node is removed). At the same time, a new edge which accidentally has the same label is introduced in R.

In the remainder of this dissertation we assume that, if the node-mapping and edge-mapping functions P_{node} and P_{edge} of $P: L \rightarrow R$ are **not** explicitly mentioned, nodes in L with a particular label are always mapped on nodes in R with the same label (if present). If $v \in V_L$ and $w \in V_R$ with $label(v)=label(w)$ then $v \in dom(P_{node})$ and $P_{node}(v)=w$. A similar assumption can be made for the edges. If $e \in E_L$ and $f \in E_R$ with $label(e)=label(f)$, $P_{node}(source(e))=source(f)$ and $P_{node}(target(e))=target(f)$ then $e \in dom(P_{edge})$ and $P_{edge}(e)=f$.

III 2.2.3 Abstract Graphs and Subgraphs

In most practical situations, we are only interested in the labels of the nodes and the edges in a graph, and not the internal representation of the nodes and the edges, given by the sets V and E, respectively. For example, in Figure 7 we do not care that the nodes are internally represented by v_1 to v_6 while the edges are represented by e_1 to e_9. The only things of importance are the labels, types and constraints of the nodes and the edges, and how the edges and nodes are connected.

Mathematically, this idea is expressed by means of **isomorphic labelled graphs**. Two (L,C)-labelled T-typed graphs G and H are isomorphic if there exists a labelled graph isomorphism between them. This means that both labelled graphs *have the same structure* modulo a renaming of their node and edge sets V and E. Because the definition is exactly the same as for ordinary graphs, it is not repeated here. In the literature, an isomorphism class of graphs is usually called an **abstract graph**.

Note that, depending on whether the category ***LTGraphL***, ***LTGraphLT*** or ***LTGraphLCT*** is used, more constraints are put on the isomorphisms. In ***LTGraphL***, only the node labels need to be the same between any two isomorphic graphs. In ***LTGraphLT***, the types too need to be the same, and in ***LTGraphLTC*** even the constraints need to be the same.

Notation. When dealing with abstract labelled graphs, the following shortcut notation is adopted. If $v, w \in V_G$ and $e \in E_G$ with $label(v)=n$, $type(v)=\omega$, $label(w)=m$, $edge(e)=(a,v,w)$ and $type(e)=\tau$, the notation $\boldsymbol{(a,n,m) \in G}$ or $\boldsymbol{(a,n,m,\tau) \in G}$ is used to denote the edge $e \in E_G$. Similarly, using the injectivity constraint on nodes, $\boldsymbol{n \in G}$ or $\boldsymbol{(n,\omega) \in G}$ is written to denote the node $v \in V_G$.

Reconsidering the example of Figure 7, $(a,\upsilon) \in G$, $(f,\omega) \in G$ and similarly for all other nodes of G. Likewise, $(p,a,b,\tau) \in G$, $(q,b,a,\phi) \in G$, and similarly for all other edges of G. Because the internal representation of the nodes and edges (i.e., the node set V_G and edge set E_G) becomes irrelevant when dealing with abstract graphs, it can be removed from Figure 7, thus obtaining a visual representation that becomes more readable, as illustrated in Figure 9. From now on, we always work with abstract graphs, unless explicitly stated otherwise.

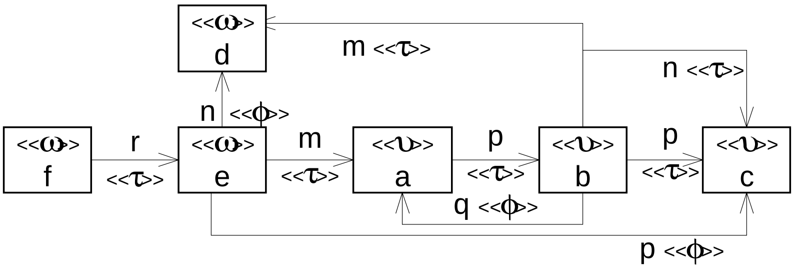


Figure 9: An abstract labelled graph

Completely similar to Definition 5, we can specify when a labelled graph *H* is a **subgraph** of a different graph *G*. It suffices to find a label and type preserving graph morphism *m: H→G*. Because the definition of labelled graphs require the node label of each node in the graph to be unique (injectivity of *vlabel*), there will be only one possible match *m: H→G*. If different nodes were allowed to have the same label, it would be possible to find more than one match of *H* into *G*.

There is one special kind of subgraph that is worthwhile mentioning. Sometimes, we are only interested in edges of a particular type τ. If this is the case, the graph can be restricted by only mentioning these edges, while hiding all others. Such a restricted graph is called a *τ-spanning subgraph*.

Let *G* be an *(L,C)*-labelled *T*-typed graph, and $\tau \in EdgeType$.
$\tau(G)$ is called the **τ-spanning subgraph** of *G* if
(1) $\tau(G)$ is a subgraph of *G* (with label-preserving and type-preserving match $m: \tau(G) \rightarrow G$)
(2) $\forall e \in E_{\tau(G)}: etype_{\tau(G)}(e)=\tau$
(3) $m_{node}: V_{\tau(G)} \rightarrow V_G$ is bijective
(4) $\forall e \in E_G$ with $etype_G(e)=\tau$: $\exists f \in E_{\tau(G)}$ with $m_{edge}(f)=e$

Definition 17: Spanning subgraph

Condition **(2)** of this definition states that $\tau(G)$ only contains edges of type τ. Condition **(3)** states that all nodes of *G* are also present in $\tau(G)$. Finally, condition **(4)** states that all edges with type τ in *G* are also contained in $\tau(G)$. As a concrete example, the ϕ-spanning subgraph of abstract graph *G* in Figure 9 is depicted in Figure 10.

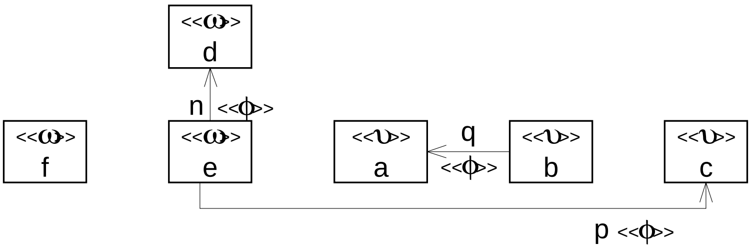


Figure 10: ϕ-spanning subgraph

III 2.3 Nested Graphs

Nesting is a natural way for humans to control the complexity of a system. In a nested graph, the overall complexity is reduced by allowing nodes to contain entire graphs themselves. In the research literature, sometimes the term *nested graphs* is used, as in [Poulovassilis&Levene94], and sometimes the term *hierarchical graphs* is preferred [Engels&Schürr95]. In this dissertation we stick to the term *nested graphs*.

In order to formally define nesting in the presence of labelled typed graphs, Definition 11 of labelled graphs (page 47) needs to be revised. In a similar way, the definition of labelled typed graphs (Definition 14) should be modified.

An ***(L,C)*-labelled nested graph** *G* is a tuple *(g, label, constraint,* ***nested****)* such that:
(1) *(g, label, constraint)* is an *(L,C)*-labelled graph.
(2) *nested:* $V \rightarrow V$ is a partial node mapping function such that its corresponding relation *nested* $\subset V \times V$ is acyclic and loop-free.
(3) *vlabel:* $V \rightarrow$ *NodeLabel* is no longer required to be injective. Instead, it needs to satisfy the following partial injectivity constraint: if *nested(v) = nested(w)* then *vlabel(v)* $\neq$ *vlabel(w)*.

Definition 18: Nested labelled graphs

Constraint **(2)** is needed to ensure that we have a *nesting hierarchy*, i.e., each node is nested in at most one other node, and there are no cycles or loops. The reason for this is that nesting is considered to be an encapsulation mechanism, and a node cannot be encapsulated in two different nodes at the same time.

Using function notation, *nested(v)* denotes the node in which *v* is nested. Using relation notation, $(v,w) \in nested$ denotes that *v* is directly nested in *w*. The transitive closure $(v,w) \in nested^+$ can be used to express that *v* is indirectly nested in *w*.

Constraint **(3)** weakens the injectivity constraint of labelled graphs. Different nodes are allowed to have the same label, as long as these nodes are not nested inside the same parent node. Otherwise, it will not be possible to distinguish them graphically. Indeed, nested graphs are represented visually by drawing nested nodes inside one another, as shown in Figure 11. If we consider *abstract* nested graphs, we have the technical problem that some nodes can have the same label. Hence they cannot be distinguished from one another by only specifying their label. This can be dealt with by using the trick of recursively qualifying nested node labels by the label of their parent node (using dot notation). For example, all nodes in the nested labelled graph of Figure 11 will be referred to (using a breadth-first order) as *A*, *A.A*, *A.B*, *A.C*, *A.A.D*, *A.A.E*, *A.C.B* and *A.C.D*, respectively. In this way, they can still be referred to in a unique way, even though some of them have the same node label.

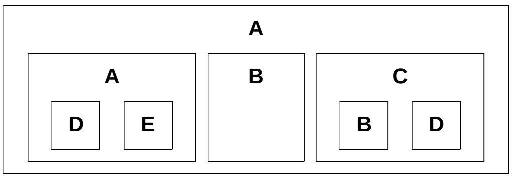


Figure 11: Visual representation of labelled nested graphs

Because we do not want to unnecessarily restrict the use of nested nodes, we allow to put an edge between any two nodes, even if these nodes are nested inside other nodes. An example of this is given in Figure 12. Because this is not always desirable in practice, additional constraints can be imposed that restrict the *nested* relation if this is necessary in the considered domain.

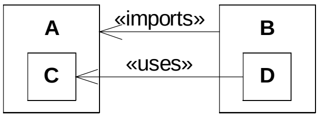


Figure 12: Edges in nested graphs

In the same way as labelled graphs and labelled typed graphs form a category (Definition 13 and Definition 16), it is possible to show that their nested variants form a category. We can even distinguish two kinds of categories based on whether or not the morphisms are required to preserve the nesting hierarchy.

Let G and H be *(L,C)*-labelled *nested* graphs.
f: G→H is a **labelled *nested* graph morphism** if it is a labelled graph morphism between the *(L,C)*-labelled graphs G and H.
A ***nesting-preserving* labelled nested graph morphism** *f: G→H* is a labelled graph morphism such that

$$\forall v \in dom(f_{node}):\ f_{node} \circ nested_G = nested_H \circ f_{node}$$

Definition 19: Nesting-preserving morphisms between labelled nested graphs

To avoid yet another new name for the category with labelled nested graphs as objects and labelled nested graph morphisms as morphism, we take the convention of using the same name as before: ***LGraph***. The category with nesting-preserving morphisms will be referred to as ***LGraphNest***.

Obviously, the definitions above can also be extended to the case of labelled typed nested graphs and their corresponding categories ***LTGraph*** and ***LTGraphNest***.

III 2.4 Types Revisited

III 2.4.1 Motivation

Although typing as introduced before is already very useful, in most practical situations we need to attach *constraints to types* as well (instead of to individual nodes and edges only). These constraints must hold for all nodes (or edges) of the corresponding type. This approach is very similar to the object-oriented approach, where all objects that are instances of the same class have the same operations and attributes, although the actual values of their attributes can differ from one object to another.

Another useful mechanism that can be borrowed from the object-oriented paradigm is the inheritance mechanism on classes. All operations of a superclass are automatically inherited by their subclasses. The formal equivalent in terms of graphs is a *subtyping mechanism* on types. All constraints specified on a particular node type or edge type are automatically inherited by all the subtypes.

III 2.4.2 Typed Graphs

In Definition 11 of page 47, constraints were introduced on **individual nodes and edges**, using the functions *vconstraint: V→℘(NodeConstraint)* and *econstraint: E→℘(EdgeConstraint)*, respectively. These constraints could be attached graphically to the node or edge between curly braces *{…}*. We will not discuss this kind of constraint in more detail here, but simply use the constraints when needed throughout this dissertation.

In Definition 14 of page 49, a labelled typed graph contained a node-typing function v*type: V→NodeType* to attach a type to each node, and an edge-typing function e*type: E→EdgeType* to attach a type to each edge. A major disadvantage of this form of typing is that it cannot be used to put constraints between edge types and the types of their source and target nodes. Typically, edges of a particular type are only allowed between nodes of a particular type. In order to express this kind of information, the types themselves should carry a graphical structure. The fixed graph T that expresses such constraints on edge types (and node types) is called the **type graph**. Its category-theoretical definition and more related definitions and properties can be found in [Corradini&al96a, Corradini&al96b, Heckel&al96]. In the presence of a type graph, Definition 15 of type-preserving morphisms remains virtually unchanged.

Let *Type = (NodeType, EdgeType)* be a pair of disjoint and finite sets of predefined types.
Let ***T*** be a fixed *(Type,C)*-labelled graph, called the **type graph**. An ***(L,C)*-labelled *T*-typed graph** is a pair *(G, type)* such that G is an *(L,C)*-labelled graph and *type: G→T* is a total ***LGraph***-morphism.
A **labelled *T*-typed graph morphism** is a labelled graph morphism *f: G→H* between *(L,C)*-labelled *T*-typed graphs.
A ***T*-preserving labelled typed graph morphism** is a labelled *T*-typed graph morphism *f: G→H* such that $type_H \circ f = type_G\ \ (\forall x \in dom(f))$

Definition 20: Typed graphs

In the above definition, node labels of the type graph T correspond to node types in an *(L,C)*-labelled *T*-typed graph. Similarly, edge labels of the type graph T correspond to edge types in an *(L,C)*-labelled *T*-

typed graph. Each labelled *T*-typed graph needs to satisfy the constraints imposed by the type graph *T*. If the type of a node or edge is changed, it needs to be checked if there are no type constraints that become invalidated.

We can redefine the categories ***LTGraph*** and ***LTGraphT*** using the revised definitions of labelled typed graph morphisms above. In the remainder of this dissertation, we always work with these revised definitions.

An immediate result of Definition 20 is that arbitrary constraints can be expressed in the type graph *T*, since it is defined as a *(Type,C)*-labelled graph. As a result, a set of constraints can be attached to each node type and edge type. Among others, this can be used to add multiplicity constraints on edges, for example to express that a node with type *«object»* can contain **at most one** edge of type *«instance»* to a node of type *«class»*.

The attentive reader may have noticed that a type graph is nothing more than a graph at meta-level. This is similar to the object-oriented analysis and design methodologies, where classes are represented in a class diagram, and objects (i.e., instances of classes) are represented in an object diagram, and the class diagram is considered to be the meta level of the object diagram. More information about approaches towards object-oriented metamodelling can be found in [Henderson-Sellers&Bulthuis98], [OMG97b] and [OMG97c]. Of course, metamodelling is not necessarily restricted to object-oriented approaches. For example, it is also used in entity-relationship diagrams [Chen76]. Interestingly, [Heckel&al96] shows how type graphs can be used to deal with entity-relationship diagrams.

III 2.4.3 Partially-Ordered Types

From a practical point of view it is also very useful to put node types and edge types in a **subtype relationship**. Similarly to the inheritance relationship in object-oriented languages and the subtype relationship in typed programming languages, this allows us to define constraints for one type in the hierarchy, and these constraints are inherited automatically by all its subtypes. Moreover, existing constraints can be overridden in subtypes, while new constraints can be added. Because we also want to allow multiple inheritance, an ordinary hierarchy does not suffice. Therefore a structure of *partial order* needs to be imposed on the node types and edge types.

A *(Type,C)*-labelled type graph *T* is **$(\leq_V,\leq_E)$-ordered** if $(NodeType, \leq_V)$ and $(EdgeType, \leq_E)$ are partial orders.

Definition 21: Partially ordered type graph

Using this definition, Definition 20 of type graphs can be extended to take partial orders into account. The partial order will be used to propagate constraints to subtypes, as specified by the following properties:

If *T* is a $(\leq_V,\leq_E)$-ordered type graph, then the following **subtype properties** must hold:

(1) $\forall v, w \in V_T$: if $vlabel(v) \leq_V vlabel(w)$ then $vconstraint(w) \subseteq vconstraint(v)$

(node constraints of supertypes are inherited by subtypes)

(2) $\forall e \in E_T$: $\forall\ s_1, s_2, t_1, t_2 \in V_T$: if $source(e)=s_1$, $target(e)=t_1$, $vlabel(s_2) \leq_V vlabel(s_1)$ and $vlabel(t_2) \leq_V vlabel(t_1)$ then $\exists f \in E_T$ with $source(f)=s_2$, $target(f)=t_2$, $elabel(f)=elabel(e)$ with $econstraint(f)=econstraint(e)$

(edge constraints between supertypes are inherited)

Definition 22: Subtype properties for a type graph

III 2.4.4 Nested Type Graph

In the presence of nesting, we do not only want to express constraints such as which types of edges can be put between which types of nodes, but we also want to specify which types of nodes can be *nested* in which other types of nodes. To achieve this, we will additionally introduce edges with label *nested* in the *(Type,C)*-labelled type graph *T*: $Type = (NodeType, EdgeType \cup \{nested\})$. As opposed to the other kinds of edge labels in the type graph *T*, *nested*-edges impose additional constraints on the relation $nested \subset V \times V$ that is defined for each *(L,C)*-labelled *T*-typed nested graph (Definition 18).

An example is given in the type graph of Figure 13 where a *nested*-edge is put from a *feature*-node to a *classifier*-node. This expresses the constraint that nodes with type *«feature»* must always be nested in nodes with type *«classifier»*:

$\forall v \in V$: if *type(v)=«feature»* then $\exists w \in V$ with *(v,w)∈nested* and *type(w)=«classifier»*

III 2.4.5 Example

As mentioned in chapter I, an important aim of the thesis is *“**to deal with software evolution in a domain-independent way**”*. In practice, this will be achieved by using a different type graph *T*, as well as a different partial order for each specific domain to which the formalism will be applied. If desired, even the labelling sets can be customised to the specific domain.

As a concrete example, consider the type graph in Figure 13, which is specifically destined to deal with *class diagrams*. For convenience, only those edges that are essential have been drawn, and not the ones that are inherited by means of the subtype properties. The partial order on node types and edge types is presented in Figure 14. It specifies that nodes can have type *interface* or *class* (which are subtypes of *classifier*), and types *operation* and *attribute* (which are subtypes of *feature*). Edges can have types *association*, *implements* and *specialises*. Edges with type *association* are only allowed between nodes of type *class*. Similarly, the source of an edge of type *implements* should always be a node of type *class*, while the target node of such an edge should always have type *interface*.

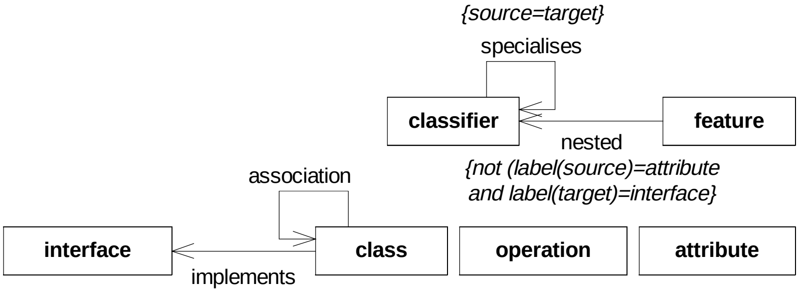


Figure 13: Type graph for class diagrams

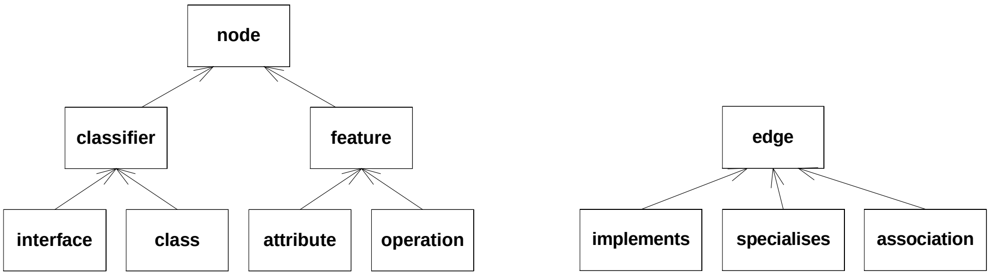


Figure 14: Node and edge type partial order

Another constraint is attached to the *specialises*-edge on node *classifier*, to state that edges with type *specialises* are only allowed between nodes of the same type, which must be of type *classifier*. Because *interface* $\leq_V$ *classifier* and *class* $\leq_V$ *classifier* (as shown in Figure 14), the subtype properties of Definition 22 guarantee that there can be *specialises* edges between *interface* and *class*, *class* and *interface*, *class* and *class*, and *interface* and *interface*. In order to restrict *«specialises»*-edges so that they can only occur between two *«interface»*-nodes or two *«class»*-nodes in a concrete graph, we have attached the additional type constraint *{source=target}* to the *specialises* edge in the type graph *T* of Figure 13. Actually, this constraint is only a shortcut notation to express the following well-formedness constraints that should hold for all typed graphs *G* that are instances of the type graph *T*:

$\forall e \in E_G$ *with type(e)=«specialises»: type(source(e)) = type(target(e))*

As explained in the previous subsection, the *nested* edge between *feature* and *classifier* expresses a constraint on nested graphs, namely that *«feature»*-nodes must always be nested in *«classifier»*-nodes. Again, the subtype properties specify that the *nested*-edge from *feature* to *classifier* is inherited by subtypes *interface* $\leq_V$ *classifier*, *class* $\leq_V$ *classifier* *attribute* $\leq_V$ *feature* and *operation* $\leq_V$ *feature*. This gives rise to four more specific nesting constraints: *«operation»*-nodes can be *nested* in *«class»*-nodes, *«operation»*-nodes can be *nested* in *«interface»*-nodes, *«attribute»*-nodes can be *nested* in *«class»*-nodes, and *«attribute»*-nodes can be *nested* in *«interface»*-nodes. To express that the latter situation is not allowed in instances of the type graph *T* of Figure 13, we have added the additional edge type constraint *{not (label(source)=attribute and label(target)=interface)}*. Again, this constraint imposes extra well-formedness restrictions on instances of the type graph *T*, and should be read as follows for each *T*-typed graph *G*:

$\forall e \in E_G$ *with type(e)=«nested»: not (type(source(e))=«attribute» and type(target(e))=«interface»)*

III 2.4.6 Subtype Preserving Morphisms

Using the structure of partial order on node types and edge types, an interesting new kind of labelled typed graph morphisms can be defined. Instead of defining type-preserving graph morphisms which must preserve types exactly, a weaker variant could be defined, where the *types of nodes and edges can be replaced by subtypes*.

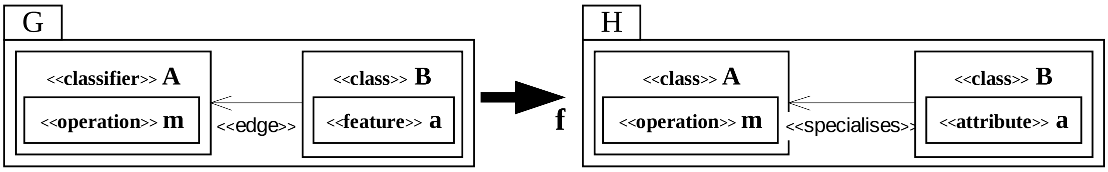


Figure 15: Subtype preserving morphism

Figure 15 shows an example of this, where both labelled *T*-typed nested graphs *G* and *H* are typed with the type graph *T* presented in Figure 13, and with partial orders as defined in Figure 14. *G* and *H* are connected by means of a label-preserving morphism $f: G \rightarrow H$. It is not type-preserving, since the type of *A* changes from *«classifier»* to *«class»*, the type of *a* changes from *«feature»* to *«attribute»*, and the type of the edge from *B* to *A* changes from *«edge»* to *«specialises»*. Nevertheless, as we can see, a type is always changed to a subtype. For this reason, we refer to this kind of morphism as a **subtype-preserving morphism**. Formally, such a morphism can be defined as follows:

Let *T* be a fixed *($\leq_V$,$\leq_E$)*-ordered *(Type,C)*-labelled graph.
Let *G* and *H* be two *(L,C)*-labelled *T*-typed graphs.
A labelled typed graph morphism $f: G \rightarrow H$ is **subtype-preserving** if

$$type_H \circ f \leq type_G$$

LTGraphT$_s$(L,C,T) is a category with *(L,C)*-labelled *T*-typed graphs as objects and subtype-preserving labelled typed graph morphisms as morphisms. Moreover, ***LTGraphT*** $\subseteq$ ***LTGraphT$_s$*** $\subseteq$ ***LTGraph***.
In a similar way, a morphism is called **super-type preserving** if $type_H \circ f \geq type_G$. This leads to a category ***LTGraphT^s*** which is the dual category of ***LTGraphT$_s$***, since the morphisms are inversed.

Definition 23: Subtype and supertype preserving morphisms

In the definition above, $type_H \circ f \leq type_G$ is actually a shortcut for

$$vtype_H \circ f_{node} \leq_V vtype_G \text{ and } etype_H \circ f_{edge} \leq_E etype_G$$

which can be expressed more precisely as

$$\forall v \in dom(f_{node}): vtype_H(f_{node}(v)) \leq_V vtype_G(v) \text{ and } \forall e \in dom(f_{edge}): etype_H(f_{edge}(e)) \leq_E etype_G(e)$$

Obviously, all labelled typed graphs must always satisfy the subtype properties of Definition 22. For example, if node *B* would have type *«interface»* instead of *«class»* in Figure 15, $f: G \rightarrow H$ would no longer be a morphism. Indeed, if *B* would have type *«interface»*, *H* would not be a valid graph, since it does not satisfy the constraints imposed by the type graph: a *«specialises»*-edge cannot go from an *«interface»*-node to a *«class»*-node (because of the type constraint *{source=target}*). Likewise, an *«attribute»*-node cannot be *nested* in an *«interface»*-node (because of the other type constraint).

III 2.4.7 Other Type Constraints

Some constraints cannot be expressed formally in a type graph. Two examples of this were presented in Figure 13, where extra type constraints (between curly braces *{…}*) were attached to the *specialises* and *nested* edge. An important kind of constraint that is also impossible to express in the type graph is the so-called **multiplicity constraint**. It states something about the *amount* of edges or nodes that are allowed in specific situations. For example, *each «interface»-node must be implemented by at least one «class»-node*. Using UML notation, this would be visually modelled in the type graph by attaching `1..*` to the source of the *implements* edge. Besides multiplicity constraints, other constraints that are impossible to express in a type graph are constraints that deal with types of more than one edge or more than two nodes at the same time.

When setting up type constraints, one should always take care that the constraints are **consistent**, i.e., they do not contradict each other or interact with each other in undesired ways. For example, when we have the following two constraints: "*a «class»-node cannot be the target of an «assoc»-edge*" and "*an «assoc»-edge can only have «class»-nodes as source and target*", the combination of these two constraints makes presence of «*assoc*»-edges impossible, which is probably not the intention.

In [Fradet&al99], a formal attempt is made to deal with the issues above. First of all, multiplicity constraints à la UML can be attached to each edge in a graph by defining two functions m_{source}*:* $E \rightarrow \boldsymbol{I} \setminus [0,0]$ and m_{target}*:* $E \rightarrow \boldsymbol{I} \setminus [0,0]$ where $\boldsymbol{I}$ is the set of nonempty intervals over $\mathbb{N}$. Using this definition, a characterisation is given of when a type graph is **consistent**, namely if there exists at least one graph that satisfies the multiplicity requirements imposed by the type graph. As it turns out, checking the consistency can be achieved by solving a system of linear inequalities!

As a second contribution, [Fradet&al99] introduces a simple constraint language which is much more powerful than the constraints that can be expressed directly in the type graph. Also for this constraint language, a consistency checking algorithm is provided.

Another constraint language which is more popular, but substantially less formal, is OCL (Object Constraint Language). It is the constraint language that is part of the UML standard [OMG97d].

III 2.5 Possible Enhancements

Although the graphs introduced in this chapter are already fairly sophisticated, their expressiveness can still be enhanced in various ways. We decided not to do this in order not to make the formalism too complex. Nevertheless, we will summarise the most interesting enhancements below:

- We could make use of *hypergraphs*, where nodes can be linked to each other by means of *hyperedges*. This is more general, because hyperedges are allowed to have more than one source and target node.
- We could provide an explicit *encapsulation* or *information hiding* mechanism on top of nested graphs. It allows to selectively hide from the outside world nodes and edges that are nested in another node. This is very useful from a practical point of view, especially when we use graphs to represent software artifacts, because information hiding is one of the essential principles of programming languages [MacLennan87]. In [Engels&Schürr95] it is explained how nested (or hierarchical) graphs can be augmented with a powerful notion of encapsulation.
- The notion of *type graphs* as defined in this chapter can be used as a *classification mechanism*, similar to the *class* concept in object-oriented programming. All graphs that have the same type graph can be considered as *instances* of this type graph. When drawing the analogy with databases, sometimes the term *graph schema* is used [Engels&Schürr95]. Because type graphs are ordinary graphs themselves, they could also have a type graph associated to them, called the *graph metaschema*. In the same way, we could go on ad infinitum and define graph meta-metaschemas, etc… This more general approach of having arbitrary levels of graph schemas is discussed in [Engels&Schürr95].
- Another interesting enhancement is to allow for *first-class edges*. Among others, this would allow for nesting at the level of edges as well, by making it possible to nest an entire graph in an edge. It would also become possible to allow edges that have an edge as source or target. The obvious disadvantage of first-class edges is that the difference between nodes and edges becomes less clear.

III . 3 GRAPH REWRITING

Until now, we have focussed on the kind of graph that will be used for modelling evolvable software artifacts. We will now take a closer look at the mechanisms that can be applied for modifying (i.e., evolving) these graphs. We have decided to apply the theory of **graph rewriting**, because it is immediately suitable for our approach, and because there is a wide spectrum of theoretical results available. After giving an overview of graph rewriting research in the first subsection, we discuss the many definitions and properties that will be needed in order to provide a formal foundation for reuse contracts in terms of graphs and graph rewriting.

III 3.1 OVERVIEW OF GRAPH REWRITING

The idea of graph grammars or graph rewrite systems is a direct generalisation of the work that has been performed on *string grammars* or *term rewrite systems*. The research area of *graph grammars* is about three decades old. The first paper about graph grammars dates from 1969, when Pfaltz and Rozenfeld [Pfaltz&Rozenfeld69] discussed many important topics concerning graph grammars, albeit in an informal way and restricted to so-called *web grammars*. Nevertheless, many of the issues reported there remain hot research topics today. For example, the proposal to extend graph production rules with *application preconditions* can already be found there. Application conditions are used to attach additional constraints to production rules. A production cannot always be applied whenever its left-hand side is found in the host graph, but only if additionally the application precondition of the production holds for this host graph. In [Montanari70], the possibility of *negative* application preconditions is indicated, meaning that a derivation step can be carried out only if some structure does *not* exist in the host graph (so-called *forbidden contexts*). Again, this work only deals with web grammars, and describes them in an informal way. More recent (and more theoretical) research on application preconditions (positive as well as negative) can be found in [Ehrig&Habel86, Habel&al96]. [Heckel&Wagner95] and [Heckel95] extend this idea to deal with application *postconditions* as well, that are used to specify constraints that must hold after applying a production to a graph.

There are many different kinds of graph grammar formalisms available. One possible distinction is between *parallel* and *sequential* graph grammars. In parallel graph grammars, a production can be applied in parallel to different subgraphs of a given graph. With sequential graph grammars, productions can only be applied one after the other, albeit in a nondeterministic way (in the sense that the subgraph to which each production is applied is chosen arbitrarily whenever there is more than one choice).

From a mathematical point of view, a distinction can be made between the *category-theoretical* (or *algebraical*) approach [Ehrig79] and the *set-theoretical* approach. In the set-theoretical approach, sets are used to represent the underlying graph structure, while in the algebraical approach categories and morphisms are used. A further distinction is made between the *single-pushout approach* [Löwe93] and the *double-pushout approach*. From a user’s point of view, the single-pushout approach differs from the double-pushout approach in as much as its productions are able to delete dangling edges. In the double-pushout approach, extra “dangling edge” and “identification” conditions are needed. For this reason, the single-pushout approach is more elegant, and it allows one to simplify many proofs.

Since graphs can be seen as a kind of data structure, the research on graph grammars can be generalised directly to the so-called *structure grammars* (or *structure rewrite systems*) [Ehrig&al91]. Instead of only being able to rewrite graphs, arbitrarily complex structures can be used. In [Löwe93] the general category of *graph structures* is used to prove many interesting properties.

One of the reasons why graph grammars are still fairly unknown in the programming community is that from its very beginning in the early 1970's the focus was on providing theoretical rather than practical results. As a result, working implementations based on these concepts were not available for a very long time. Fortunately, this is beginning to change. For example, a very powerful graph grammar based visual programming language called PROGRES (PROgramming with Graph REwriting Systems) has been developed [Schürr95]. Actually, PROGRES is only one specific instance of the field of *programmed graph rewriting*. Programmed graph rewriting systems use imperative control structures for regulating the application of rewrite rules. Although unnecessary from a theoretical point of view, programmed graph grammars are very practical when using graph grammars in real-world situations, e.g., for specification purposes.

Nevertheless there are still some unavoidable efficiency problems related to graph grammars, partly because looking for subgraphs in a given graph is relatively time-consuming. Currently, some work is going on to make graph grammar implementations more efficient, but this is outside the scope of this dissertation.

III 3.2 Basic Definitions

From now on, the word *graph* is used consistently instead of *(L,C)-labelled T-typed graph*, because all the definitions presented here are valid for many different kinds of graph-like structures such as ordinary graphs (labelled or unlabelled), hypergraphs, as well as more complex structures. In order to provide a general framework to prove interesting results, [Löwe93] works in the so-called category of *graph structures*, which encompasses all previously mentioned cases. This category has the important property that it is closed with restriction to finite colimits. As a result, pushout constructions, as for example needed in Definition 26 below, are guaranteed to exist. Many interesting results have been shown for these graph structures, and we will choose the ones that are most appropriate for application in the domain of reuse contracts.

In [Corradini&al96a], it is shown that the definitions and properties for ordinary graph grammars can be translated directly to corresponding definitions and properties about typed graph grammars.

Most of the definitions given in this section are taken from [Löwe93], sometimes with a slightly different terminology.

III 3.2.1 Productions and Derivations

Our approach will be to model an evolvable software artifact as a graph rewriting system. Starting from an initial graph (the evolvable artifact), a set of primitive productions (evolution operations) can be used to transform (evolve) the graph into a new evolved version.

Essentially, a **graph rewriting system** consists of a set of initial graphs and a set of productions.

A **graph rewriting system** *GG* is a pair *(SG,SP)* such that
(1) *SG* is a set of initial graphs
(2) *SP* is a set of productions

Definition 24: Graph rewriting system

Remarks:

- In many situations, a *grammar* is used instead of a *rewriting system*. Although these terms are often used as synonyms for each other, the main distinction is that a grammar is a set of productions that generates a language of terminal graphs and produces non-terminal graphs as intermediate results. A rewriting system on the other hand, is a set of rules that transforms one instance of a given class of graphs into a different instance of the same class of graphs without distinguishing terminal and non-terminal results.
- Usually, only one graph is chosen as initial graph in a graph rewriting system. For convenience however, we also allow one to start from a finite set of initial graphs.

A **production** *P* basically consists of a left-hand side *L*, a right-hand side *R*, and an embedding transformation *embed*. In order to apply a production *P* to a given graph *G*, an occurrence of its left-hand side *L* in *G* is replaced by the right-hand side *R*, while *embed* specifies the details of how the right-hand side should be inserted or embedded. In other words, it specifies what needs to be done with the edges of *G* that arrive in or leave from nodes of *L*, once *L* has been replaced by *R*. The kind of embedding transformation that is used is an important criterion of distinction between the different graph rewriting approaches. *Embed* does not necessarily preserve node labels. It can even be an empty function, meaning that all edges from outside *L* arriving or leaving in *L* will be removed in the new graph. In this dissertation, the trivial approach is taken, where all edges adjacent to a node of *L* will still be adjacent to the corresponding node of *R* (if present) designated by the *embed* function. If there is no corresponding node in *R*, the edges will be removed from the graph.

Let *L* and *R* be graphs. A **production** *P: L→R* is a partial graph morphism from the left-hand side *L* to the right-hand side *R*.

Definition 25: Algebraical definition of a production

This definition corresponds to the *algebraical approac*h to graph transformations. Essentially, the algebraical approach attempts to describe graph rewriting using graph morphisms and gluing constructions for graphs as basic concepts for the construction of derivations. In this formalism, most definitions can be given in an elegant and abstract way. As a result, many proofs can be reduced significantly, and results can be shown simultaneously for an entire range of different structures. On the other hand, however, the abstractness of most definitions makes them more difficult to understand. More importantly, the fact that they do not bother about operational details makes it more difficult to apply them to results in actual tools and software applications. For this reason, an operational definition remains desirable in some cases. For those readers not familiar with category theory, an intuitive explanations for the used concepts is provided as much as possible.

Using productions, **direct derivations** can be defined as the application of a production $P: L \rightarrow R$ to a given graph G.

Let G and H be graphs, and $P: L \rightarrow R$ a production.
$G \Rightarrow_{P,m} H$ is a (**direct**) **derivation** (or an **application of *P* to *G***) if $\exists$ total graph morphism $m: L \rightarrow G$ that forms a **pushout** with $P: L \rightarrow R$.

Definition 26: Direct derivation

In the appendix of [Löwe93], an operational variant of this definition is formally specified, and it is shown that this operational definition is equivalent to its algebraic variant.

The subgraph L of G is called an **occurrence of *L* in *G***, or a **match** for production P (hence the name $m: L \rightarrow G$). As illustrated in Figure 16 and defined in Definition 73 of page 213, the **pushout** construction implies that existence of a derived graph H and graph morphisms $\boldsymbol{m^*: R \rightarrow H}$ (called the **co-match of *m***) and $P^*: G \rightarrow H$ that make the diagram commute. Because of this, any direct derivation is not only determined by its production P, but also by the corresponding match $m: L \rightarrow G$. This is why the notation $G \Rightarrow_{P,m} H$ is used. In the practical examples of the next chapter, however, the match m is often omitted when it is clear from the context. Note that the pushout construction ensures that H is the *minimal* graph that can be obtained by applying P to G (by means of match m).

$$\begin{tikzcd}
L \arrow[r, "P"'] \arrow[d, "m"] & R \arrow[d, dashed, "m^*"] \\
G \arrow[r, dashed, "P^*"'] & H
\end{tikzcd}$$

Figure 16: Single Pushout construction for direct derivations

Using the notion of (direct) derivation, a *derivation sequence* can be defined as a sequence of direct derivations.

Let $G_0, \dots G_n$ be graphs, and $P_1, P_2, \dots P_n$ productions with matches $m_1, \dots, m_n$. $\boldsymbol{G_0 \Rightarrow^+ G_n}$ is a **derivation sequence** if a sequence of direct derivations of the form $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} \dots \Rightarrow_{Pn,mn} G_n$ exists. We write $\boldsymbol{G_0 \Rightarrow^* G_n}$ if $G_0 = G_n$ or $G_0 \Rightarrow^+ G_n$

Definition 27: Derivation sequence

Making use of derivation sequences, we can also define the **graph language** corresponding to a graph rewriting system:

Let $GG = (SG, SP)$ be a graph rewriting system. Its **graph language** $L(GG)$ is defined as the (possibly infinite) set of all graphs that can be derived directly or indirectly from one of the initial graphs. More precisely:

$$L(GG) = \{ G \mid Start \in SG \text{ and } Start \Rightarrow^* G \}$$

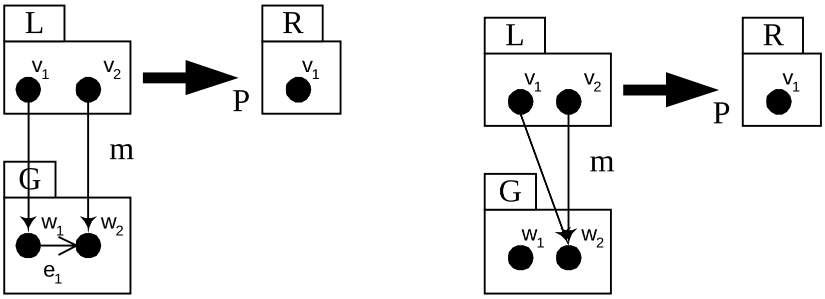
Definition 28: Graph language

III 3.2.2 Conflicts when Applying a Production

While Definition 26 presents the so-called *single-pushout approach* (SPO) towards graph transformations, there is an alternative which is also frequently used, namely the *double-pushout approach* (DPO). Without diving in details, the basic difference between them is that a production P in the DPO is based on **two total morphisms** $l: K \rightarrow L$ and $r: K \rightarrow R$ (instead of one partial morphism $P: L \rightarrow R$), where K represents the *gluing graph* into which added components are going to be integrated.

We prefer the SPO over the DPO because the productions $P: L \rightarrow R$ are significantly simpler. This substantially reduces many proofs. Moreover, it is shown in [Löwe93] that the SPO is actually a generalisation of the DPO, implying that all existing results are still valid in the new framework!

Because of the simplicity of the productions, however, some derivations may have unintuitive side-effects due to the absence of restrictions on the match $m: L \rightarrow G$. As illustrated in Figure 17, there are two kinds of conflicts that can occur when applying a production $P: L \rightarrow R$ to a graph G by means of a match $m: L \rightarrow G$.



Dangling edge conflict *Identification conflict*

Figure 17: Conflict between preservation and deletion of u

To illustrate the first conflict, assume we have a production $P: L \rightarrow R$ that removes node v_1 while preserving node v_2. On the left of Figure 17, an injective match $m: L \rightarrow G$ specifies that L is a subgraph of G which contains an extra edge e_1 between the two nodes. Because one of these nodes is removed by P, a **dangling edge conflict** arises. In the SPO, this results in a graph H containing only one node w_1. While, in the DPO it is easy to avoid dangling edge conflicts by specifying that a node can be replaced/removed only when there is no adjacent edge, this is *not* the case for the SPO. To overcome these problems, the SPO can be extended with *application conditions* which impose extra preconditions (on L) and postconditions (on R). This will be explained in section III 3.5.

The second kind of conflict illustrated on the right of Figure 17 occurs when the match $m: L \rightarrow G$ identifies nodes v_1 and v_2 into w_2. As a result, v_2 should be preserved as well as deleted, which is called an **identification conflict**. Again the SPO will yield a graph H containing only one node w_1 in this case. In order to avoid this second kind of conflict, the extra requirement of **conflict-freeness** can be added to a production.

Let $m: L \rightarrow G$ be a match for production $P: L \rightarrow R$. m is **conflict-free for *P*** if $m(x)=m(y)$ implies either $x, y \in dom(P)$ or $x, y \notin dom(P)$.

Definition 29: Conflict freeness

Note that this definition of conflict freeness only avoids the *identification conflict*, but not the *dangling edge conflict*. If m is an injective match, it is always conflict-free (because of the definition of injectivity: $m(x)=m(y)$ implies $x=y$).

If $m: L \rightarrow G$ is injective, then m is conflict-free for any $P: L \rightarrow R$.
If $m: L \rightarrow G$ is total and conflict-free for P, then its co-match m^* is total.
If P is injective, then P^* is injective.

Property 3: Some pushout properties

III 3.3 Parallel and Sequential Independence

This section investigates under which conditions parallelly independent derivations of the same graph can be serialised. *In the context of software evolution, it is very important to know this, since it corresponds to the merge or combination of two independent evolutions of the same original software artifact!* As it turns out, there a lot of interesting properties that can help us to deal with this problem. For example, the so-called *confluence property* (or *Church-Rosser property*) states that, if two derivations are parallelly independent, they can be serialised in any order, and always lead to the same result.

All the definitions and properties in this section are taken from [Löwe93], where the results are proved using a category-theoretical approach. [Habel&al96] and [Heckel95] even go a step further by proving the results for **conditional** graph grammars (that will be presented in section III 3.4).

In the remainder of this section, we will assume that $P_1: L_1 \rightarrow R_1$ and $P_2: L_2 \rightarrow R_2$ are two arbitrary productions. Likewise, G, G_0, G_1, G_2 and H represent arbitrary graphs.

We will start with defining **parallel independence** of direct derivations. Intuitively, two direct derivations (starting from the same initial graph G_0) are parallelly independent if they can be applied in any order with the same result. The formal definition makes use of the weaker notion of **weakly parallel independence**, which states that the first derivation does not delete or introduce nodes or edges needed by the second.

Let $G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P2,m2} G_2$ be two direct derivations with corresponding matches $m_1: L_1 \rightarrow G_0$ and $m_2: L_2 \rightarrow G_0$.
$G_0 \Rightarrow_{P2,m2} G_2$ is **weakly parallelly independent** of $G_0 \Rightarrow_{P1,m1} G_1$ iff the occurrence of L_2 in G_0 is preserved in G_1 by P_1, i.e., $m = P_1^* \circ m_2: L_2 \rightarrow G_1$ is a match for P_2.
$G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P2,m2} G_2$ are **parallelly independent** if they are mutually weakly parallelly independent.

Definition 30: Parallel independence of direct derivations

A schematic representation of weakly parallel independence is given on the right of Figure 18. If m makes the diagram commute, P_1 and P_2 are weakly parallelly independent. A similar definition can be given for determining sequential independence of two direct derivations that have been applied one after the other. Informally, a derivation is **weakly sequentially independent** of a preceding derivation if it could already have been applied before that. In other words, the second derivation should not rely on elements newly generated by the first. **Sequential independence** additionally requires that the second derivation does not delete anything needed by the first. Its graphical representation is presented on the left of Figure 18. If a match m exists that makes the diagram commute, P_1 and P_2 are weakly sequentially independent.

Let $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} G_2$ be a derivation sequence of two direct derivations with corresponding matches $m_1: L_1 \rightarrow G_0$ and $m_2: L_2 \rightarrow G_1$.
$G_1 \Rightarrow_{P2,m2} G_2$ is **weakly sequentially independent** of $G_0 \Rightarrow_{P1,m1} G_1$ if the occurrence of L_2 in G_1 was also present in G_0 and is preserved by P_1, i.e., $\exists$ match $m: L_2 \rightarrow G_0$ such that $P_1^* \circ m = m_2$
$G_0 \Rightarrow_{P1,m1} G_1$ and $G_1 \Rightarrow_{P2,m2} G_2$ are **sequentially independent** iff $G_1 \Rightarrow_{P2,m2} G_2$ is weakly sequentially independent of $G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P1,m1} G_1$ is weakly parallelly independent of $G_0 \Rightarrow_{P2,m} H$ (where H is obtained by applying P_2 directly to G_0 instead of G_1 by means of the match $m: L_2 \rightarrow G_0$). Under these conditions, $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} G_2$ is called a **sequentially independent derivation sequence**.

Definition 31: Sequential independence of direct derivations

$$\begin{tikzcd}
 & L_2 \arrow[r, "P_2"] \arrow[dl, dashed, "m?"'] \arrow[d, "m_2"] & R_2 \arrow[d, "m_2*"] \\
\mathbf{G_0} \arrow[r, "P_1*"'] & \mathbf{G_1} \arrow[r, "P_2*"'] & \mathbf{G_2}
\end{tikzcd}
\qquad
\begin{tikzcd}
L_2 \arrow[r, "m_2"'] \arrow[rr, dashed, bend left, "m?"] \arrow[d, "P_2"'] & \mathbf{G_0} \arrow[r, "P_1*"'] \arrow[d, "P_2*"] & \mathbf{G_1} \\
R_2 \arrow[r, "m_2*"'] & \mathbf{G_2} &
\end{tikzcd}$$

Figure 18: Weakly sequential and parallel independence

Using these definitions, we can now give the ***confluence property*** (or *Church-Rosser property*) that was mentioned at the beginning of this section. This property is very important, since it states the existence and uniqueness of a derivation sequence in the case of parallelly independent direct derivations, as well as the uniqueness of a result irrespective of the application order of the direct derivations in a sequentially independent derivation sequence. Among others, this can be used to determine under which conditions two direct derivations in a sequence are commutative. Finally, the property says that parallelly independent derivations can always be transformed into a sequentially independent derivation and vice versa. In the next chapter, this property will be used heavily in the context of software evolution.

(1) If $G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P2,m2} G_2$ are parallelly independent direct derivations, then $\exists$ graph H and direct derivations $G_1 \Rightarrow_{P2,n2} H$ and $G_2 \Rightarrow_{P1,n1} H$ such that both $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,n2} H$ and $G_0 \Rightarrow_{P2,m2} G_2 \Rightarrow_{P1,n1} H$ are sequentially independent derivation sequences.
(2) If $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} H$ is a sequentially independent derivation sequence, then $\exists$ graph G_2 and matches n_1, n_2 such that $G_0 \Rightarrow_{P2,n2} G_2 \Rightarrow_{P1,n1} H$ is a sequentially independent derivation sequence.
(3) If P_1 and P_2 are injective graph morphisms, then there is a *unique* correspondence between parallelly independent derivations $G_0 \Rightarrow_{P1,m1} G_1$, $G_0 \Rightarrow_{P2,m2} G_2$ and sequentially independent derivation sequences $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,n2} H$ resp. $G_0 \Rightarrow_{P2,m2} G_2 \Rightarrow_{P1,n1} H$ defined by $n_2 = P_1^* \circ m_2$ and $n_1 = P_2^* \circ m_1$.

Property 4: Local confluence property

Proof: See [Löwe93]. It suffices to know that in part **(3)**, the result graph H is obtained by applying the so-called *pushout* construct.

While **(1)** and **(2)** in the confluence property specify *existence* of graph H and graph G_2, respectively, **(3)** specifies *uniqueness* under the extra condition that P_1 and P_2 are injective. Fortunately, these injectivity constraints will be valid for the primitive productions that will be needed for dealing with evolution in the next chapter.

Truly parallel production applications essentially require to abstract away from any possible application order, which implies not to generate any intermediate graph. In other words, a truly parallel production application must be modelled by the application of a single production, called the **parallel production**.

The **parallel production P_1+P_2**: $L_1+L_2 \rightarrow R_1+R_2$ is obtained as the disjoint union of P_1: $L_1 \rightarrow R_1$ and P_2: $L_2 \rightarrow R_2$, as defined in [Löwe93]. $G_0 \Rightarrow_{P1+P2,\, m} H$ is called a **parallel derivation**.
$G_0 \Rightarrow_{P1+P2,\, m} H$ is called a **parallelly independent derivation** if $G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P2,m2} G_2$ are parallelly independent (with $m_1 = m \circ i_1$ and $m_2 = m \circ i_2$, where i_1: $L_1 \rightarrow L_1+L_2$ and i_2: $L_2 \rightarrow L_1+L_2$ are inclusions).

Definition 32: Parallel derivations

Based on this definition and the above confluence property, the important *parallelism property* can be proved. It states that a sequentially independent derivation sequence can be *synthesised* or *composed* into a single direct derivation that is obtained by performing all productions of the sequence in parallel. This is important for efficiency, since the derivations to not have to be applied one at a time. Instead they can be applied simultaneously. The inverse process of *composition* (or *synthesis*) is *decomposition* (or *analysis*). It states that the effect of a parallel derivation can be simulated by sequential derivations with the components of the parallel production. Moreover, the order of the derivations in the sequence is irrelevant. Again, uniqueness of the decomposition step is only guaranteed if P_1 and P_2 are injective.

Composition. If $G \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} H$ is a sequentially independent derivation sequence then $\exists$ synthesis leading to a parallelly independent derivation $G \Rightarrow_{P1+P2,\, m} H$.
Decomposition. If $G \Rightarrow_{P1+P2,\, m} H$ is a parallelly independent derivation, then $\exists$ analysis into a sequentially independent derivation sequence $G \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} H$.
Uniqueness. If P_1 and P_2 are injective, then the synthesis and analysis are unique. The unique correspondence is given by $m = m_1+n_2$ and $m_2 = P_1^* \circ n_2$.

Property 5: Parallelism of sequentially independent derivations

The proof of this property is based on Property 4 and a weaker variant of the parallelism property that is stated below:

$\exists$ parallelly independent derivation $G \Rightarrow_{P1+P2,\, m} H$ such that $G \Rightarrow_{P2,m2} G_2$ is weakly parallelly independent of $G \Rightarrow_{P1,m1} G_1$ iff $\exists$ weakly sequentially independent derivation sequence $G \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,n2} H$

Property 6: Weak parallelism

III 3.4 Composite Productions

In practice, to be able to deal with *complex evolution steps*, they will need to be modelled as sequences of usually sequentially *dependent* derivations. As a result of this sequential dependence, the previous properties cannot be used. Another problem is that, when expressing a complex evolution step as a sequence of many primitive derivations, many intermediate graphs need to be calculated before the actual result can be achieved.

To solve both problems, **composite productions** need to be introduced as productions that capture the effect of a number of (possibly sequentially dependent) productions applied one after the other. Applying a composite production to a graph leads to a **composite derivation**. The effect of this application is the same as applying each of the more primitive productions consecutively, starting from this graph. Note that, in [Heckel95] the term *derived rule* is used instead of composite production. Apart from some differences in terminology, all the results presented here are taken directly from [Heckel95].

The practical relevance of composite productions and derivations is obvious. By using a composite derivation as a shortcut for an entire derivation sequence, the number of intermediate graphs is reduced, and the interaction of the productions applied in the sequence is fixed. In this way, a semantics is obtained that is very similar to database transactions. The composite derivation behaves as an atomic action. Either all derivations in the sequence are performed together (when the composite production is applied) or they leave the system in its current state (when the composite production is not applicable). Due to the absence of intermediate states, the transaction may not be interrupted by other actions. More work on these so-called graph grammar transactions is described in [Schürr96]. There, graph grammar transactions are shown to represent recursively defined, partial, and nondeterministic graph rewriting programs.

Let P_1: $L_1 \rightarrow R_1$ and P_2: $L_2 \rightarrow R_2$ be two productions with $R_1=L_2$. Their **sequential composition** ***$P_1;P_2$***: $L_1 \rightarrow R_2$ is defined by $P_2 \circ P_1$ in the category of graph morphisms.

Definition 33: Sequential composition

Let P_1 and P_2 be defined as above.
$\exists$ **sequentially composed derivation** $G_0 \Rightarrow_{P1;P2,\, m1} G_2$ with match m_1: $L_1 \rightarrow G_0$ that is conflict-free for P_1
iff $\exists$ derivation sequence $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} G_2$ such that m_2: $L_2 \rightarrow G_1$ is the co-match of m_1: $L_1 \rightarrow G_0$ (i.e., $m_2 = m_1$*).

Property 7: Sequentially composed derivations

Definition 33 and Property 7 are illustrated graphically in Figure 19. The requirement of conflict-freeness in Property 7 is needed to avoid those cases where m_1* is partial. Indeed, if the total morphism m_1: $L \rightarrow G$ is conflict-free, its co-match m_1* will be total as well according to Property 3. An alternative approach would be to deal with partial application conditions in a consistent way.

$$\begin{tikzcd}
L_1 \arrow[r, "P_1"'] \arrow[d, "m_1"] \arrow[rr, bend left, "P_1;P_2"] & R_1=L_2 \arrow[r, "P_2"'] \arrow[d, "m_1*=m_2"] & R_2 \arrow[d, "m_2*"] \\
\mathbf{G_0} \arrow[r, "P_1*"] \arrow[rr, bend right, "(P_1;P_2)*"'] & \mathbf{G_1} \arrow[r, "P_2*"] & \mathbf{G_2}
\end{tikzcd}$$

Figure 19: Sequential composition

In practice, the constraint that $R_1=L_2$ in Definition 33 is fairly restrictive. In order to extend this definition to deal with productions that do not satisfy this constraint, these productions have to be embedded into a common context before they can be composed. This requires the notion of **derived production**.

Let $m: L \rightarrow G$ be a match for production $P: L \rightarrow R$. Then $P^*: G \rightarrow H$, generated by the pushout of P and m is called a **derived production**.

Definition 34: Derived production

Let $P^*: G \rightarrow H$ be a derived production (generated by $P: L \rightarrow R$ and $m: L \rightarrow G$).
$\exists$ **derived derivation** $K \Rightarrow_{P^*, e} Y$ with match $e: G \rightarrow K$
iff $\exists$ direct derivation $K \Rightarrow_{P, e \circ m} Y$

Property 8: Derived derivation

Using both previous results, i.e., derived productions and sequential composition, the general notion of **composite productions** and **composite derivations** can be defined.

Let $\forall i \in \{1,..,n\}$: $m_i: L_i \rightarrow G_{i-1}$ be matches for productions $P_i: L_i \rightarrow R_i$. Then $P_1^*;P_2^*;\ldots;P_n^*: G_0 \rightarrow G_n$ defined by $P_n^* \circ \ldots \circ P_2^* \circ P_1^*$ is called a **composite production**.

Definition 35: Composite production

Let $P_1, P_2, \ldots P_n$ be defined as above.
$\exists$ **composite derivation** $K \Rightarrow_{P1^*;P2^*;\ldots;Pn^*\ ,\ e} Y$ with match $e: G_0 \rightarrow K$ that is conflict-free for P_i $\forall i \in \{1,..,n\}$
iff $\exists$ derivation sequence $G_0 \Rightarrow_{P1,\ e1 \circ m1} G_1 \Rightarrow_{P2,\ e2 \circ m2} \ldots \Rightarrow_{Pn,\ en \circ mn} G_n$ with matches $e_i \circ m_i: L_i \rightarrow G_{i-1}$ such that $e_1 = e$ and e_i is the co-match of e w.r.t. P_i.

Property 9: Composite derivation

III 3.5 Conditional Graph Rewriting

III 3.5.1 Motivation

Subsection III 3.2.2 mentioned the dangling edge conflict that arises in the single pushout approach towards graph rewriting. In order to avoid this conflict, the productions need to be made more expressive by introducing so-called *application conditions*. From a practical point of view, this increased expressiveness also allows to specify many graph derivations in a more concise and more understandable way.

Application conditions for graph rewriting are introduced in [Ehrig&Habel86, Heckel95, Habel&al96]. The major contribution of conditional graph rewriting is the introduction of *negative application conditions*. Positive application conditions were already indirectly present in the ordinary graph rewriting approach. Indeed, [Habel&al96] have shown that a graph production with only positive application conditions can always be reduced to an ordinary graph production by performing a context enlargement. When taking negative application conditions into account as well, the formalism becomes more expressive. [Habel&al96] have shown that context-free graph grammars with negative application preconditions are more powerful than ordinary context-free graph grammars, while still remaining less expressive than context-sensitive graph grammars. The main contribution of [Heckel95] was to augment the formalism of [Habel&al96] with application *post*conditions. Again, this extension makes the formalism more expressive. Below, we repeat the definitions and properties needed for our purposes.

A **conditional graph rewriting** is a graph rewriting where each production $P: L \rightarrow R$ contains a set of **application conditions**. These application conditions can be subdivided in application **preconditions** and application **postconditions**. Application preconditions express constraints that should be satisfied in order for the production to be applicable to a graph. Application postconditions should be satisfied in the result graph after the production has been applied.

A further distinction can be made between positive and negative application conditions. **Positive** application conditions represent positive constraints or obligations on L (or R), and require the presence of particular nodes or edges for the production to be applicable. **Negative** application conditions represent negative constraints or prohibitions on L (or R), and require the absence of some nodes or edges. Stated otherwise, positive conditions correspond to a *required context*, while negative conditions correspond to a *prohibited context*.

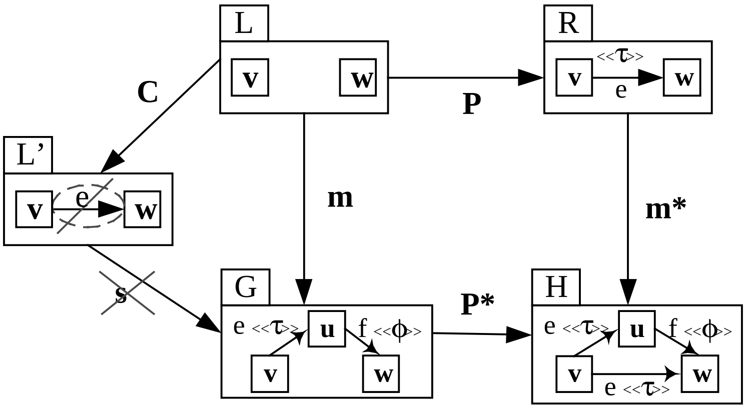


Figure 20: A negative application precondition

In Figure 20, an example is given of production $P: L \rightarrow R$ that is only applicable if a particular negative application precondition on L is satisfied. The production P adds an edge with label e and type *«τ»* between two nodes v and w, but only if there is not already such an edge present between these nodes. This constraint is specified in terms of a morphism $C: L \rightarrow L'$. When applying the production P to a graph G by means of a match $m: L \rightarrow G$, it should be impossible to find a morphism $s: L' \rightarrow G$ that makes the diagram commute. If we restrict ourselves to label-preserving morphisms, this is indeed the case, so P can be applied to G, yielding a result graph H, since the negative precondition is satisfied.

III 3.5.2 Application Conditions

Formally, application conditions are defined as follows:

Let L and G be graphs, and $m: L \rightarrow G$ a total graph morphism (i.e., a *match* of L in G).
An **application condition** $C \in ApplCond(L)$ is a total graph morphism $C: L \rightarrow L'$ (for some L').
$ApplCond(L) = ApplCond^{+}(L) \oplus ApplCond^{-}(L)$, i.e., an application condition can be positive or negative.
A **positive application condition** $C \in ApplCond^{+}(L)$ is **satisfied in *G* by *m*** if $\exists$ total graph morphism $s: L' \rightarrow G$ such that $s \circ C = m$. A **negative application condition** $C \in ApplCond^{-}(L)$ **is satisfied in *G* by *m*** if $\nexists$ total graph morphism $s: L' \rightarrow G$ such that $s \circ C = m$.
A set of application conditions $ApplCond(L)$ is **consistent** if $\exists$ graph G and $\exists$ match $m: L \rightarrow G$ such that $\forall C \in ApplCond(L)$: C is satisfied in G.

Definition 36: Application conditions

Graphically, satisfaction of a positive application condition means that the diagram of Figure 21 commutes, while satisfaction of a negative condition means that no such commuting diagram exists. Intuitively, $s: L' \rightarrow G$ can be considered as an *extended match* in G in the sense that it coincides with the match m for all elements of L, and adds some extra requirements (application conditions) that must also be matched in G.

$$\begin{tikzcd}
L \arrow[r, "m"] \arrow[d, "C"] & G \\
L' \arrow[ur, "s"'] &
\end{tikzcd}$$

Figure 21: Satisfaction of an application condition

Application conditions can either be defined as constraints on a graph directly, or as conditions on a graph production. In the first case, we speak of a **conditional graph** (Definition 37), and in the second case we speak of a **conditional production** (Definition 38).

A **conditional graph** is a triple $(G, m: L \rightarrow G, ApplCond(L))$ such that G is a graph, $m: L \rightarrow G$ is a match of L in G, and $ApplCond(L)$ is a set of application conditions that are satisfied in G by m.

Definition 37: Conditional graphs

In the case of conditional productions, a distinction can be made between application **preconditions** and application **postconditions** [Heckel&Wagner95, Heckel95]. Preconditions represent constraints on L, the left-hand side of the production $P: L \rightarrow R$. Therefore, they are sometimes called *left-sided constraints*. Postconditions represent constraints on the right-hand side R of the production P. Therefore, they are sometimes called *right-sided constraints*. *Satisfaction* of a **conditional derivation** $G \Rightarrow_P H$ means that all preconditions and all postconditions must be satisfied.

A **conditional graph rewriting system** CG is a graph rewriting system (SG, SP) such that SP is a set of conditional productions.

A **conditional production** is a pair $(P: L \rightarrow R, ApplCond_P)$ such that $P: L \rightarrow R$ is an (ordinary) production, and $ApplCond_P = ApplCond_P(L) \oplus ApplCond_P(R)$ is the set of application **preconditions** (representing constraints on L) and application **postconditions** (representing constraints on R).

Let $(P: L \rightarrow R, ApplCond_P)$ be a conditional production, and $G \Rightarrow_{P,m} H$ a direct derivation with corresponding match $m: L \rightarrow G$ and co-match $m^*: R \rightarrow H$.

An application **precondition** $C \in ApplCond_P(L)$ is **satisfied in** $\boldsymbol{G \Rightarrow_{P,m} H}$ if it is satisfied in G by m.

An application **postcondition** $C \in ApplCond_P(R)$ is **satisfied in** $\boldsymbol{G \Rightarrow_{P,m} H}$ if it is satisfied in H by m^*. $\boldsymbol{G \Rightarrow_{P,m} H}$ **satisfies** $\boldsymbol{ApplCond_P}$ if $\forall C \in ApplCond_P$: C is satisfied in $G \Rightarrow_{P,m} H$.

$G \Rightarrow_{P,m} H$ is a **conditional direct derivation** if it is a direct derivation that satisfies $ApplCond_P$.

$G \Rightarrow^* H$ is a **conditional derivation sequence** if it is a derivation sequence of conditional direct derivations. In other words, all application conditions for its direct derivations should be satisfied. It some of the conditions are not satisfied, we speak of an **invalid derivation sequence**.

Definition 38: Conditional productions and conditional derivations

Notation. We write $\boldsymbol{ApplCond_G}$ to indicate that application conditions belong to a particular graph G. We write $\boldsymbol{ApplCond_P}$ to indicate that application conditions belong to a particular production P. We usually write ***PreCond(P)*** instead of $ApplCond_P(L)$ to stress the fact that we deal with application **preconditions** of a production $P: L \rightarrow R$. We usually write ***PostCond(P)*** instead of $ApplCond_P(R)$ to stress the fact that we deal with application **postconditions** of a production $P: L \rightarrow R$.

If $(G, ApplCond_G)$ is a conditional graph, then $ApplCond_G$ is consistent.

If $G \Rightarrow_{P,m} H$ is a conditional direct derivation, then $PreCond(P)$ and $PostCond(P)$ are consistent.

Property 10: Consistency

Proof:

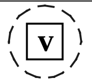
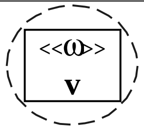
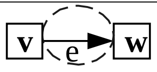
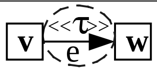
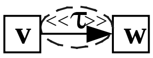
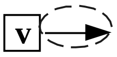
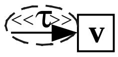
> If $(G, ApplCond_G)$ is a conditional graph, then $\forall C \in ApplCond_G$: C is satisfied in G. Hence, by definition, $ApplCond_G$ is consistent.
>
> If $G \Rightarrow_{P,m} H$ is a conditional direct derivation, then $G \Rightarrow_{P,m} H$ satisfies $ApplCond_P$. Hence $\forall C \in ApplCond_P$: C is satisfied in $G \Rightarrow_{P,m} H$. Consequently, $\forall C \in ApplCond_P(L)$: C is satisfied in G, and $\forall C \in ApplCond_P(R)$: C is satisfied in H. Hence, $ApplCond_P(L)$ and $ApplCond_P(R)$ are consistent.

The next subsections will categorise the different kinds of application conditions that will be used in this dissertation, together with how they are represented graphically and textually.

III 3.5.3 Positive Application Conditions

Basically, a positive application condition $C \in ApplCond^+_P$ corresponds to a positive constraint on the left-hand side L (precondition) or right-hand side R (postcondition) of production $P: L \rightarrow R$. In the case of a precondition, it demands that specific items should be *present* in L in order for the production to be applicable. In the case of a postcondition, it demands that specific items should be present in R after the production is applied.

There are two different alternatives for representing application conditions: using a graphical notation (such as the one introduced in [Habel&al96]), or using a textual notation. Although the graphical variant is more intuitive, it is relatively space consuming. Therefore, we prefer to present the application conditions in a **textual variant** in this dissertation. Both variants are presented below, but only for the specific case of *labelled typed graphs* (as defined on page 53) since this is the kind of graph that will be used in this dissertation.

Graphical	Textual	Explanation
	$v \in G$	Required presence of a node with label *v*
	$(v,\omega) \in G$	Required presence of a node with label *v* and type *ω*
	$(e,v,w) \in G$	Required presence of an edge with label *e* between nodes *v* and *w*
	$(e,v,w,\tau) \in G$	Required presence of an edge with label *e* and type *τ* between nodes *v* and *w*
	$\exists e \in EdgeLabel: (e,v,w,\tau) \in G$ abbreviated to $\boldsymbol{\tau(v,w) \in G}$	Required presence of at least one edge with type *τ* between nodes *v* and *w*
	$\exists e \in EdgeLabel:$ $\exists w \in NodeLabel: (e,v,w) \in G$ or shorter $\boldsymbol{OutNode_G(v) \neq \varnothing}$	Required presence of at least one edge with source node *v*
	$\exists e \in EdgeLabel:$ $\exists w \in NodeLabel: (e,w,v,\tau) \in G$	Required presence of at least one edge with type *τ* and target node *v*

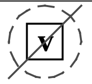
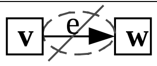
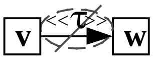
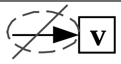
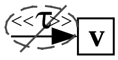
Note that the notation $(v,\omega) \in G$ can be split up into $v \in G$ and $type_G(v)=\omega$. Similarly, the notation $(e,v,w,\tau) \in G$ can be split up into $(e,v,w) \in G$ and $type_G(e,v,w)=\tau$. The latter notation will be more convenient in the proofs of some properties in the next chapter.

Also note that the constraints expressed above are not the only ones imaginable. Others can be defined in a similar way.

III 3.5.4 Negative Application Conditions

Negative application conditions $C \in ApplCond^{--}_P$ correspond to negative constraints on the left-hand side *L* (precondition) or right-hand side *R* (postcondition) of production $P: L \rightarrow R$, and demand that specific items should be *absent*.

The textual and graphical notation is very similar to the one for positive application conditions. Again only a representative selection of negative conditions is expressed below.

Graphical	Textual	Explanation
	$v \notin G$	Prohibited presence of a node with label *v*
	$(e,v,w) \notin G$	Prohibited presence of the unique edge with label *e* between nodes *v* and *w*
	$\nexists e \in EdgeLabel: (e,v,w,\tau) \in G$ abbreviated to $\tau(v,w) \notin G$	Prohibited presence of all edges with type *τ* between nodes *v* and *w*
	$\nexists e \in EdgeLabel$: $\nexists w \in NodeLabel: (e,v,w) \in G$ or shorter $InNode_G(v)=\varnothing$	Prohibited presence of all edges with target node *v*
	$\nexists e \in EdgeLabel:$ $\nexists w \in NodeLabel: (e,w,v,\tau) \in G$	Prohibited presence of all edges with type *τ* and target node *v*

Notation. As can be seen in the last but one constraint above, $\boldsymbol{InNode_G(v)=\varnothing}$ is used to state that a node *v* contains no incoming edges. Similarly $\boldsymbol{OutNode_G(v)=\varnothing}$ is used to state that a node *v* contains no outgoing edges. $\boldsymbol{Adjacent_G(v)=\varnothing}$ is used to express the last two application conditions simultaneously.

An important result about postconditions, shown in [Heckel95], is that postconditions can be transformed into equivalent preconditions. In this situation the postcondition is said to be **anticipated** by a precondition. Because of this property, postconditions can already be checked *before* the production has been applied, which is very useful for efficiency reasons.

Let $(P\colon L \rightarrow R, ApplCond_P)$ be a conditional production and $m\colon L \rightarrow G$ a match of L in G.
$\forall C\colon R \rightarrow R' \in PostCond(P)$: $\exists\, \boldsymbol{\alpha_P(C)}\colon L \rightarrow L'$ called the **anticipation of *C* along *P*** such that
$\forall G \Rightarrow_{P,m} H$: $\alpha_P(C)$ is satisfied in $G \Rightarrow_{P,m} H$ iff C is satisfied in $G \Rightarrow_{P,m} H$

Property 11: Anticipation of a postcondition by a precondition

III 3.5.5 Interesting Properties

This subsection revisits the properties and definitions that have already been discussed for unconditional graph rewriting, and sees which changes are required to cope with *conditional* graph rewriting. Fortunately, most of the results remain valid, with only some slight modifications. For more details, we refer to [Heckel95, Habel&al96].

To be valid in the context of conditional graph rewriting, the definitions of **weakly parallel independence** (Definition 30) and **weakly sequential independence** (Definition 31) need to be modified slightly, because not only the matches, but also the application conditions need to be preserved. To show the differences clearly, they have been underlined in the definition below.

Let $G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P2,m2} G_2$ be two conditional direct derivations with corresponding matches $m_1\colon L_1 \rightarrow G_0$ and $m_2\colon L_2 \rightarrow G_0$. $G_0 \Rightarrow_{P2,m2} G_2$ is **weakly parallelly independent** of $G_0 \Rightarrow_{P1,m1} G_1$ iff $m = P_1^* \circ m_2\colon L_2 \rightarrow G_1$ is a match for P_2 and $\underline{G_1 \Rightarrow_{P2,m} G_2 \text{ is a conditional direct derivation, i.e., } ApplCond_{P2} \text{ is preserved by } P_1 \text{ in } G_1.}$
Let $G_0 \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,m2} G_2$ be a conditional derivation sequence with corresponding matches $m_1\colon L_1 \rightarrow G_0$ and $m_2\colon L_2 \rightarrow G_1$. $G_1 \Rightarrow_{P2,m2} G_2$ is **weakly sequentially independent** of $G_0 \Rightarrow_{P1,m1} G_1$ iff $\exists$ match $m\colon L_2 \rightarrow G_0$ such that $P_1^* \circ m = m_2$ and $\underline{G_0 \Rightarrow_{P2,m2} H \text{ is a conditional direct derivation, i.e., } ApplCond_{P2} \text{ is preserved by } P_1 \text{ in } G_0.}$

Definition 39: Conditional parallel and sequential independence

An important property based on this definition, namely **local confluence** (Property 4) still remains valid in the case of conditional derivations.

When only considering application *pre*conditions, the results about **parallel productions** and **parallel derivations** remain valid as well in the conditional case. The definition of a conditional parallel production P_1+P_2 is basically the same as before, and the **parallelism property** (Property 5) and its weak variant (Property 6) of the unconditional case still hold. When taking application postconditions into account as well, things become more difficult, since postconditions can be used to control the iteration between productions that are applied in parallel. Because of this, particular sequential derivations can no longer be parallelised and vice versa [Heckel95]. As a result, postconditions are strictly more powerful than preconditions if parallel derivations are allowed. For our specific purposes, the problem mentioned above will not arise, since we do not really need this parallelism property.

The results for **sequential composition**, **derived productions** and **composite productions** still hold for the conditional case, but we need to take the application conditions into account. For example, if $(P_1\colon L_1 \rightarrow R_1, ApplCond_{P1})$ and $(P_2\colon L_2 \rightarrow R_2, ApplCond_{P2})$ are two conditional productions with $R_1 = L_2$, their sequential composition $P_1;P_2\colon L_1 \rightarrow R_2$ can be defined in the same way as for the unconditional case, with additionally $PostCond(P_1;P_2) = PostCond(P_2)$, and $PreCond(P_1;P_2)$ is obtained from $PreCond(P_1)$ and by anticipating all postconditions of P_1 and all preconditions of P_2. In a similar way, derived productions can be defined by also embedding the application conditions in a common context. Finally, composite conditional productions can be defined by combining the previous two results. Moreover, Property 7, Property 8 and Property 9 remain valid.

III 3.5.6 Consistency Conditions

Instead of using pre- and postconditions, that must be specified for each production separately, an alternative approach is to make use of so-called *consistency conditions* [Heckel&Wagner95]. These are basic graph properties that must be preserved by the graph productions. In other words, they correspond to *graph invariants*. Formally, consistency conditions are defined exactly the same as application conditions.

Let CC be a set of application conditions, called **consistency conditions**. A conditional graph rewriting system *(SG, SP)* is **consistent with respect to *CC*** if
$\forall G \in SG$: $\forall c \in CC$: c is satisfied in G.
$\forall P \in SP$: $\forall c \in CC$: if $G \Rightarrow_{P,m} H$ and c is satisfied in G then c is satisfied in H.

Definition 40: Consistency conditions

The consistency conditions of [Heckel&Wagner95] have been shown to be transformable into pre- and postconditions for individual productions. On the other hand, the proposed consistency conditions are restricted in the sense that they can only express very basic conditions such as existence or uniqueness of nodes and edges. More structural conditions like, e.g., existence of cycles, cannot be expressed in their approach.

III 3.6 Possible Enhancements

The conditional graph rewriting formalism presented above can be extended in different ways. One possible approach is to use *propositional application conditions* that allow one to combine constraints by means of conjunction, disjunction and negation. Another alternative is to express *cardinality conditions* that describe upper of lower bounds for the in- or out-degree of a given node. Both approaches have already been investigated to some extent in [Heckel95]. Another track which has been investigated there is the use of *partial application conditions*, to deal with cases where the match m is not conflict-free, in which case the co-match m^* is not total.

The graph rewriting formalism defined in this chapter is still fairly primitive, and can be extended in various ways to enhance its expressiveness. Below we discuss some interesting extensions to our formalism based on PROGRES [Schürr95] (an acronym for PROgramming with Graph REwriting Systems). This is a graph-grammar-based programming language that tries to close the gap between the “operation-oriented” manipulation of graphs by means of rewrite rules, and the “declaration-oriented” description of graphs by means of logic-based languages. In this way, both disciplines - graph grammar theory and logic theory - are able to profit from each other. PROGRES also takes the pragmatic approach of intermixing visual and textual notations. Visual constructs are used for defining subgraph tests, while textual control structures are used for the declaration of paths and restrictions. Since the visual style of programming is more intuitive, the textual style of programming should only be used when necessary, e.g., when transitive closures are needed.

By looking at the features offered in PROGRES, we can immediately find a number of useful extensions:

- Because application conditions are only defined to work on single nodes, there is sometimes a need for extending them to deal with *sets of nodes*. An even further distinction is made between *required* and *optional* nodes. Required nodes are required for the rewrite rule to be applicable, while optional nodes are not.
- Often needed paths in a given graph can be specified in PROGRES by means of *path expressions*. These path expressions can then be used as complex application conditions for graph rewrite rules.
- In order to facilitate reuse of components, it should be possible to express them in a generic way, by making use of some kind of *template mechanism*. In PROGRES, limited support for this exists. More specifically, node classes can be defined as a kind of template node. Properties that are valid for node classes are also valid for all its instances. In this way, *generic graph rewrite rules* (also called *parameterised productions*) can be defined that have nodes as parameters. Note that, despite their need, edge parameters are not supported in PROGRES due to unsolved type checking problems.

III . 4 SUMMARY

This chapter presented an underlying domain-independent formalism of graphs and graph rewriting. In the next chapter, a formal foundation for software evolution (based on reuse contracts) will be defined on top of this formalism. Graphs will correspond to evolvable software artifacts, while graph rewriting will be used to express evolution of these artifacts.

The graphs that have been introduced in this chapter have the following important features:

- They contain a *label* and a *constraint set* for each node and edge in the graph.
- They contain a *nesting mechanism* that can be used to reduce the overall complexity, by allowing entire graphs to be nested inside nodes.
- They contain a *typing mechanism* on nodes and edges. Type constraints can be expressed in a type graph, or by using ordinary constraints.
- There is a *subtyping relationship* defined on node types as well as edge types.

The latter two features are very useful when customising the domain-independent formalism to a specific domain. It suffices to specify the domain-specific node types and edge types, express the constraints that should be satisfied by these types, and determine the subtyping relationships.

The graph rewriting formalism that was chosen allowed us to rely on many useful definitions and powerful properties:

- The *single pushout approach* towards graph rewriting was adopted. This is a very powerful, though fairly abstract, way of dealing with *graph productions* and *graph derivations*.
- *Parallel independence* allowed us to specify when two derivations of the same graph were serialisable. The *confluence property* said that, in the case of two parallelly independent productions, the order in which they are serialised is irrelevant, and always leads to the same result graph.
- To deal with sequences of sequentially dependent productions, the notion of *composite production* and *composite derivation* was used. This allowed us to consider a derivation sequence as an atomic whole.
- To increase the expressiveness of graph rewriting, application conditions were introduced. *Preconditions* could be used to express constraints that need to be satisfied before a production can be applied, while *postconditions* need to be satisfied after application of the production. A further distinction was made between positive and negative application conditions. Moreover, it turned out that nearly all results of the unconditional case were immediately generalisable to the conditional case.

IV . A Formal Foundation for Reuse Contracts

This chapter presents a formal foundation for software evolution based on reuse contracts. This foundation is defined on top of the underlying formalisms of graphs and graph rewriting. We explain how it can be used to detect undesired interactions when software artifacts evolve.

IV . 1 INTRODUCTION

In this chapter, a formal foundation for reuse contracts is proposed, using the underlying formalism of labelled typed graphs and conditional graph rewriting introduced in the previous chapter. More specifically, we explain how reusable software artifacts can be expressed by means of labelled typed graphs, and how reuse and evolution of these artifacts can be expressed by means of conditional graph rewriting.

IV 1.1 MERGING INDEPENDENTLY EVOLVED COMPONENTS

An essential aspect of reuse contracts is that they allow us to render reuse and evolution more disciplined by identifying different kinds of modification. Each kind of modification is specified by a *contract type*. These types impose obligations, permissions and prohibitions on the provider and the reuser. Contract types are fundamental to disciplined reuse and evolution, as they form the basis for detecting conflicts when evolving components are reused. In this chapter, we will see that *conditional productions* provide a natural way for defining contract types.

A *reuse contract* will be defined as a *conditional graph derivation*, i.e., the application of a conditional production (read: contract type) to an initial graph (read: provider clause). When comparing two different reuse contracts that have the same provider clause, i.e., two different derivations that modify the same initial graph (as abstractly represented in Figure 22) they sometimes make conflicting modifications. An introductory example was already presented in Figure 5 (section II 5.6), with *G_0 = WebNavigation*, *G_1 = HistoryNavigation* and *G_2 = PDFNavigation*.

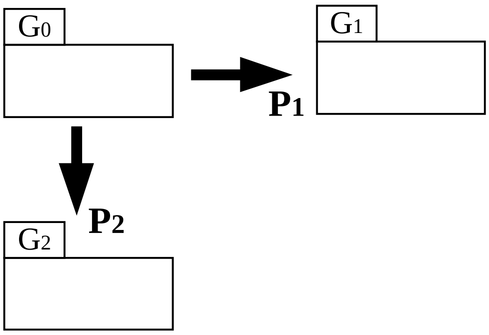


Figure 22: Detecting evolution conflicts between primitive reuse contracts

In practice, the ability to detect potentially undesired interactions between parallel evolutions of the same software artifact is very important. It is especially needed in the context of collaborative software development, where software can be modified simultaneously by different developers, usually for different reasons (different bug fixes, implementing different functionalities, etc…). When merging the modifications made by these developers, unforeseen problems often turn up.

The first thing that needs to be investigated is in which cases these problems arise. By employing a primitive set of contract types, it is possible to identify all possible ways in which two modifications can interact. Two kinds of undesired interactions will be distinguished: *structural* or *syntactic inconsistencies*, and *behavioural* or *dynamic inconsistencies*.

IV 1.2 DETECTING INCONSISTENCIES BETWEEN PARALLEL EVOLUTIONS

Structural or *syntactic inconsistencies* occur when two independent evolutions of the same artifact cannot be merged because a part of the component, which is required for the first evolution, is modified by the second evolver (or vice versa). Because of this, the first evolution can no longer be applied. Formally, this problem will be referred to as an *applicability conflict*. A typical example of such a conflict occurs when one (conditional) production removes a node from a graph, while a different (conditional) production adds a new edge from this node to an other node. In section IV . 3 , a formal definition and categorisation of the various kinds of applicability conflicts will be presented. The formal definition is based on the notion of *parallel independence* of (conditional) productions. If two productions are *parallelly dependent* they cannot be merged. If they are parallelly *independent*, the local confluence property of the previous chapter (Property 4) guarantees that they can be serialised, giving rise to a new unique result graph that can be considered as the merge or combination of the two independently evolved components.

Even if two evolutions of the same component are parallel independent, their sequential merge might still inadvertently give rise to undesired interactions. These so-called *behavioural* or *semantic inconsistencies* are harder to detect from a graph rewriting point of view. Indeed, there are several cases where two independently evolved versions of the same component can be merged, but where the resulting combination does not exhibit the desired behaviour according to one or both evolvers. This is the case when implicit assumptions made by this evolver are broken by the other evolver. For this reason, we speak of *evolution conflicts*. A number of different ways in which these evolution conflicts can be detected are presented in section IV . 4 . Because undesired interactions usually occur because of undocumented assumptions that are breached during evolution, one can never be sure if there is a real problem or not. Therefore, we take a “worst case” approach, by detecting all potentially undesired interactions.

Before we can start considering applicability or evolution conflicts, we must start by investigating *primitive contract types* in section IV . 2 . These primitive contract types represent the basic modifications that can be made to a graph. Together, they can describe any possible graph modification. In subsection IV 2.1 we start by giving a motivating example. Section IV 2.2 formally defines the primitive contract types in terms of conditional productions.

IV . 2 Defining Reuse Contracts Formally

IV 2.1 Labelled Typed Graph Productions

This section shows how the approach of reuse contracts fits in the underlying formalism of labelled typed graph rewriting presented in the previous chapter.

IV 2.1.1 Decisions

In order to represent software artefacts, we will use *(L,C)*-labelled *T*-typed graphs, where *T* is a fixed type graph, *L* a fixed labelling set and *C* a fixed (but possibly infinite) set of constraints. In other words, from now on we work in the category ***LTGraph***, or one of its subcategories.

We will also make a second restriction, because the graph rewriting formalism defined in the previous chapter is too general for our needs. In general, the result of applying a (conditional) production to a graph is not necessarily unique. If $G \Rightarrow_P H$ is a graph derivation corresponding to a production $P: L \rightarrow R$, *G* can contain more than one subgraph which the left-hand side *L* of *P* can be matched to. For each of these matches, a different result graph *H* will be generated.

In this chapter, however, we will restrict ourselves to injective (total) matches $m: L \rightarrow G$ and injective productions $P: L \rightarrow R$. In many cases, we will even make use of label-preserving matches and productions (***LTGraphL***-morphisms), which are mostly type-preserving as well (***LTGraphLT***-morphisms).

Because of the injectivity requirement, applying the (conditional) production $P: L \rightarrow R$ to *G* will always lead to a unique result graph *H*. Since the productions are injective, the uniqueness condition of Property 4 (local confluence) and Property 5 (parallelism) is also satisfied. Finally, by using Property 3 of page 61, we know that the derivation $G \Rightarrow_P H$ is injective (since *P* is injective), and that m^* is total (since *m* is injective and total).

For some specific purposes, we need matches $m: L \rightarrow G$ that are injective but not label-preserving. For example, to detect evolution conflicts we need to check if a particular graph pattern *L* is present in a particular graph *G*. Because a graph pattern only needs to take the node types and edge types into account, and since the node labels and edge labels are not known in advance, the match does not need to be label-preserving. Whenever we encounter such a situation, we will explicitly mention the fact that labels are not necessarily preserved. Formally, this can be achieved by working in the category ***LTGraph***.

IV 2.1.2 Example

To illustrate what a derivation looks like when working with labelled typed nested graphs, consider the example in Figure 23. Although it already involves nesting for the sake of the presentation, we will postpone the formal treatment of evolution in the presence of nesting until chapter V, and deal with the basic issues first.

The example illustrates a “refactoring” production $P: L \rightarrow R$, where a common parent is introduced for two nodes with more or less the same information. The left-hand side *L* contains a *Circle* and *Triangle* node of type *«object»*, which both have subnodes *area* and *perimeter* of type *«attribute»*. Additionally, both objects are the source of a *center* edge with type *«has-a»* and target-node *Point* of type *«object»*. The production *P* transforms this configuration by introducing a new node *Geo* of type *«object»* which is the parent of *Circle* and *Triangle* (by means of two *«is-a»* edges with empty label). The subnodes *perimeter* and *area*, as well as the *center* edge, are moved to this parent-node *Geo*. In this way, redundancy is removed, and the design is made more reusable.

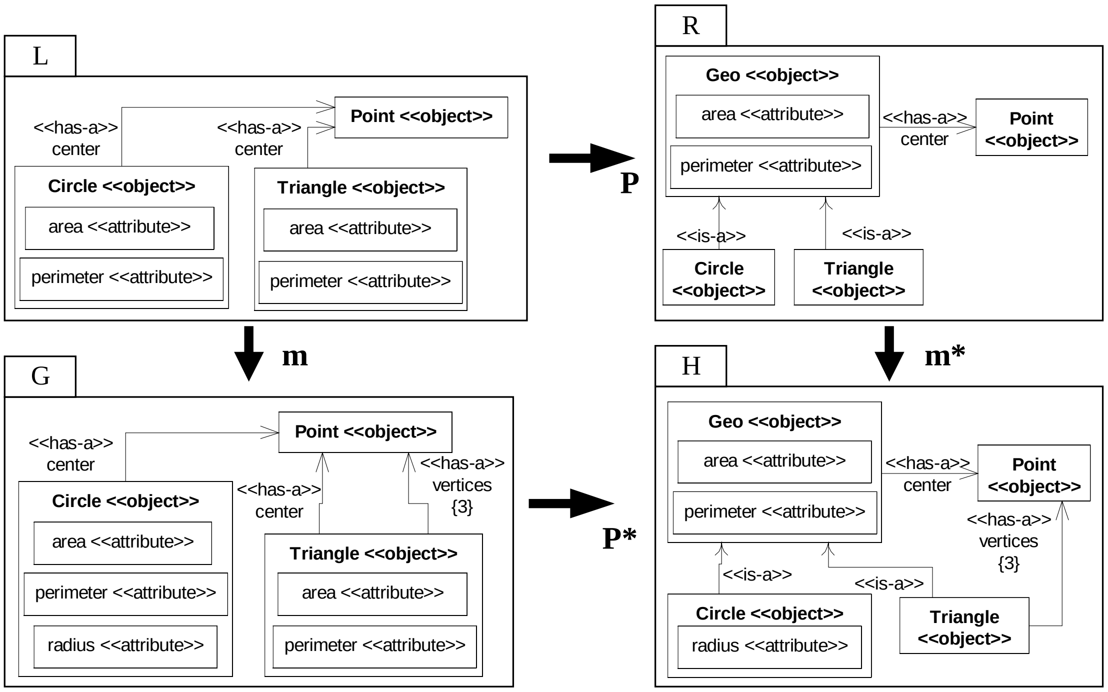


Figure 23: A derivation example

While the production *P: L→R* expresses the essential changes that need to be made to perform the refactoring, the left-hand side *L* is actually part of a larger graph *G* in which the *Circle*-node contains an additional *radius*-subnode, while the *Triangle*-node is the source of an additional *«has-a»*-edge with label *vertices* and constraint *{3}* (expressing that each triangle contains 3 points as vertices). The relationship between *L* and *G* is specified by an injective match *m: L→G*, which is a **total *LTGraphLT***-morphism, since it preserves labels and types, and is defined for each node and edge of *L*. In this specific example, the morphism *P: L→R* is an ***LTGraphLT***-morphism, although in general we do not require that the types are preserved (so *P* can sometimes be an ***LTGraphL***-morphism). Moreover, *P: L→R* is a **partial *LTGraph***-morphism, since it is not defined for all nodes and edges of *L*. More specifically, the subnodes *area* and *perimeter* of *Circle* and *Triangle* do not have a counterpart in *R*, and similarly for the edges *(center,Circle,Point)* and *(center,Triangle,Point)*.

When applying the production *P* to *G*, we obtain the result graph *H*, which contains the effect of applying the refactoring in the context of *G*. Some caution is needed if there are *identification* or *dangling edge* conflicts (as defined in section III 3.2.2). Because *m* is injective, we do not have to worry about any identification conflicts, according to Property 3. A dangling edge conflict, however, can still occur. This would be the case, for example, if there was a *«uses»*-edge between the *perimeter* and *radius* subnodes of *Circle* in *G*. Indeed, applying *P* to *G* would result in a dangling *«uses»*-edge to *radius*, since the *perimeter* node has been removed by *P*.

Fortunately, by attaching appropriate negative preconditions to the productions, and by only considering a significant subset of primitive productions, this kind of dangling edge conflict too can be avoided. More specifically, a dangling edge conflict can be avoided by only allowing the removal of a node *v* from a graph *G* by means of a production *P: L→R* if there are no adjacent edges in *v*. In a similar way as we have shown in Figure 20, this can be expressed by a negative precondition *C: L→L'* that prohibits the presence of nodes to or from *v* in *G*.

We will later see that dangling edge conflicts can still appear in a different situation, where the same graph is modified by different productions, and an applicability conflict occurs when trying to merge these productions.

IV 2.1.3 Reuse Contracts

From the example above we can derive how various parts of a reuse contract (as informally defined in section II 5.3) can be specified formally using the theory of graphs and graph rewriting. Obviously, a *provider clause* corresponds to a labelled typed graph *G*. The kind of modification that takes place is specified by a *contract type*, which corresponds to a graph production *P: L→R*, i.e., a partial ***LTGraph***-

morphism. The *modifier clause*, which specifies the exact details of the modification, can be defined by means of a total match *m: L→G*, that identifies exactly how the contract type *P: L→R* will be applied to the provider clause *G*. Intuitively, the modifier clause can be obtained by filling in the parameters of the contract type, or by "instantiating" the contract type in the context of the provider clause. For simplicity, we require the match *m* to be label-preserving as well as type-preserving (i.e., an ***LTGraphLT***-morphism). Finally, the *reuse contract* itself is defined by the graph derivation *P*: G→H*, which is obtained by the pushout of *P: L→R* and *m: L→G*.

In the next section we will start by identifying the primitive contract types we will use.

IV 2.2 PRIMITIVE CONTRACT TYPES

A labelled typed graph can be modified (during evolution or reuse) in many different ways. Each modification corresponds to a (conditional) graph rewriting production. To reduce the complexity of the problem, we will split up all the possible modifications that can be made to a graph into a small set of elementary productions. These elementary productions will correspond *to primitive contract types* in reuse contract terminology. By restricting ourselves to this set of primitive contract types, and using only these productions for expressing evolution, it will become possible to give a complete characterisation of the applicability and evolution conflicts that arise when different persons modify the same graph.

Compared to the reuse operators of [Lucas97] and [Steyaert&al96] our primitive contract types will be defined in a more orthogonal way. As a result, we will be able to combine these primitive contract types into composite ones, while still able to detect the same evolution conflicts.

In this section, we will not yet look at contract types that involve nesting or subtyping. In section V . 6 , we will address the necessary changes to cope with these scalability issues.

IV 2.2.1 Definitions

Basically, the label-preserving modifications that can be performed on a labelled typed graph are: adding a node or edge to a graph, their inverses of removing a node or edge from a graph, and changing the type of a node or edge. These six elementary modifications are called *Extension*, *Refinement*, *Cancellation*, *Coarsening*, *NodeRetyping* and *EdgeRetyping*, respectively. Except for the latter two, these modifications (or slight variations thereof) were already proposed in [Steyaert&al96], but in a less general way.

The latter two modifications are not type-preserving, so they correspond to ***LTGraphL***-morphisms. The first four modifications can be seen as label-preserving and type-preserving morphisms in the subcategory ***LTGraphLT***. This leads us to the following definition of primitive contract types and primitive reuse contracts:

A conditional production *(P: L→R, ApplCond$_P$)* with *P* an ***LTGraph***-morphism is a **primitive contract type** if *P* ∈ { *Extension(v,ω), Cancellation(v,ω), Refinement(e,v,w,τ), Coarsening(e,v,w,τ), NodeRetyping(v,υ,ω), EdgeRetyping(e,v,w,τ,φ)* } where *v, w* ∈ *NodeLabel, υ, ω* ∈ *NodeType, e* ∈ *EdgeLabel, τ, φ* ∈ *EdgeType*.
Let *G* and *H* be two graphs, and *m: L→G* a total ***LTGraphL***-morphism. A **primitive reuse contract** (generated by the pushout *P, m*) is a conditional direct derivation of the form $G \Rightarrow_{P,m} H$ where *P* is a primitive contract type.

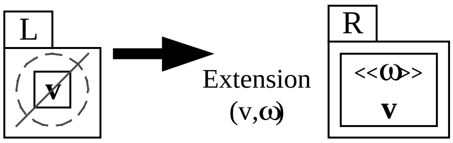
Definition 41: Primitive contract types and reuse contracts

The exact definitions of the six primitive contract types are given below. For convenience, we have represented all negative application preconditions in a single graph using the notation introduced in section III 3.5. We do not explicitly mention *positive* preconditions, since these can be expressed in *L* immediately. Moreover, we do not mention the application *post*conditions here, since these can be anticipated by preconditions. Finally, whenever the types of the nodes are not relevant in the production, they will not be mentioned in the graphical representation.

Extension

Description. Add a new node to the graph.

Definition: *∀v ∈ NodeLabel: ∀ω ∈ NodeType:*



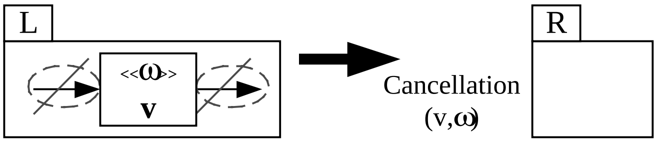
∈ ***LTGraphLT***

Remark. The *Extension* contract type needs to be parameterised (or *instantiated*) with a node label and node type before it can be applied. For example, *Extension(Geo,«object»)* adds a new node with label *Geo* and type *«object»* to the graph if there is not yet a node with label *Geo* present in the graph. The labels and edges of all other nodes and edges are preserved.

Cancellation

Description. Remove a node from the graph.

Definition: *∀v ∈ NodeLabel: ∀ω ∈ NodeType:*



∈ ***LTGraphLT***

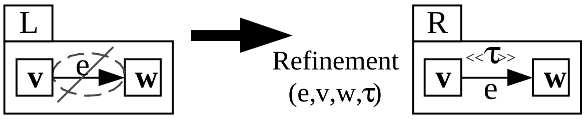
Remark. *Cancellation* can only be applied if the node with label *v* (and type *ω*) that needs to be removed contains no incoming or outgoing edges, i.e., *Adjacent(v)=∅*. The *Cancellation* contract type needs to be parameterised with the label and type of the node that needs to be removed. For example, *Cancellation(area,«attribute»)* removes the node with label *area* and type *«attribute»* from the graph.

Cancellation is the only contract type that is allowed to remove nodes. Hence, it is the only production that can give rise to a *dangling edge* conflict (as defined in section III 3.2.2). However, because of the imposed negative preconditions, such a conflict can never arise, since a *Cancellation* is not applicable if there are still incoming edges to or outgoing edges from the node that needs to be cancelled.

Refinement

Description. Add a new edge to the graph.

Definition: *∀v, w ∈ NodeLabel: ∀e ∈ EdgeLabel: ∀τ ∈ EdgeType:*



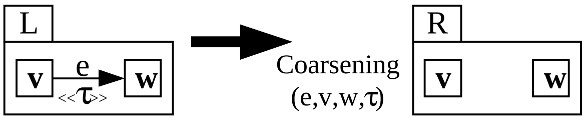
∈ ***LTGraphLT***

Remark. To apply a *Refinement*, the label and type of the edge are required, as well as the labels of the source and target nodes between which the edge should be added. For example, *Refinement(center,Geo,Point,«has-a»)* will add an edge with label *center* and type *«has-a»* to the graph if the nodes *Geo* and *Point* already exist, while an edge with label *center* does not yet exist between those nodes.

Coarsening

Description. Remove an existing edge from the graph.

Definition: *∀v, w ∈ NodeLabel: ∀e ∈ EdgeLabel: ∀τ ∈ EdgeType:*

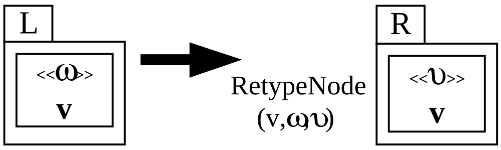


∈ ***LTGraphLT***

Remark. To apply a *Coarsening*, the label and type of the edge as well as the label of its source and target nodes are required. For example, *Coarsening(center,Circle,Point,«has-a»)* removes the edge with label *center* and type *«has-a»* between the nodes *Circle* and *Point*.

NodeRetyping

Description. Change the type of a node in the graph.
Definition: $\forall v \in NodeLabel\text{: } \forall \omega, \upsilon \in NodeType \text{ with } \upsilon \neq \omega$:

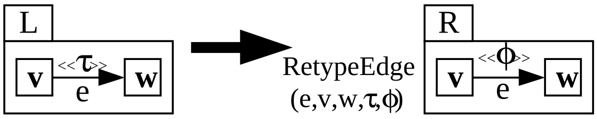


∈ ***LTGraphL***

Remark. To apply a *NodeRetyping*, the label of the node, as well as the original and new type are required, and the new type should differ from the original one. For example, *NodeRetyping(A,«abstract»,«concrete»)* changes the type of *A* from «*abstract*» to «*concrete*». The labels and types of all other nodes and edges are preserved.

EdgeRetyping

Description. Change the type of an edge in the graph.
Definition: $\forall v, w \in NodeLabel\text{: } \forall \tau, \phi \in EdgeType \text{ with } \tau \neq \phi$:



∈ ***LTGraphL***

Remark. To apply an *EdgeRetyping*, the label, source and target node of the edge are required, as well as its original and new type. Again, the new type should differ from the original one.

In order to define a primitive reuse contract $G \Rightarrow_P H$ based on the primitive contract types $P: L \rightarrow R$ above, we need to find a match $m: L \rightarrow G \in$ ***LTGraphL***. This match should always be label-preserving (and hence injective), and is only type-preserving if the type of the node or edge is explicitly mentioned in the figure above. In other words, for *Refinement(e,v,w,τ)*, *Coarsening(e,v,w,τ)* and *EdgeRetyping(e,v,w,τ,φ)*, the match $m: L \rightarrow G$ does not need to preserve the type of the nodes *v* and *w*.

Because all the primitive contract types above are label-preserving, they correspond to injective morphisms, as formalised in the following property:

A primitive contract type is an injective graph morphism.

Property 12: Injectivity of primitive contract types

Proof:

If *P* is a primitive contract type, then $P \in$ ***LTGraphL***. Hence *P* is injective, according to Property 2 of page 48.

IV 2.2.2 Discussion of the Primitive Contract Types

When considering a software system that is still under development, *Extension* and *Refinement* will be the most common forms of evolution. Indeed, if we use an incremental object-oriented approach, new requirements are gradually implemented by adding new classes, objects, attributes and operations, and by adding relationships between them.

After a while, however, it might be the case that existing classes, operations, relationships etc… become obsolete, and need to be removed from the system. In that case, *Cancellation* and *Coarsening* are needed. Another example where these contract types are needed is to make a software system more efficient. For example, instead of accessing a variable indirectly through its accessor function, one could replace this by directly updating the variable. As a result, the accessor function can be removed from the system by means of a *Cancellation*. One should note that this kind of modification is not very desirable, since it makes the system less reusable. However, since it does occur in practice, we need to provide a mechanism to cope with it in a disciplined way.

The last two primitive contract types, *NodeRetyping* and *EdgeRetyping*, are used less frequently. Nevertheless, they are indispensable during software development. They arise often when an existing system is being “refactored” or “restructured” to enhance its reusability and maintainability [Opdyke92, Opdyke&Johnson93, Johnson&Opdyke93]. For example, [Johnson&Opdyke93] explains that in some specific situations it is useful to convert a generalisation relationship into an aggregation relationship (or vice versa). In our approach, this would be achieved by performing an *EdgeRetyping*, where the type of the edge under consideration is changed from *«generalisation»* to *«aggregation»* or vice versa. Another example where retypings are useful is when a general kind of element or relationship is "refined" into a more specific one. For example, an abstract class or operation can be made concrete, or an association relationship can be changed into a composition or aggregation relationship (which can be considered as a special kind of association).

IV 2.2.3 Orthogonality Issues

The definitions of the four primitive contract types *Extension*, *Refinement*, *Cancellation* and *Coarsening* were carefully chosen, such that they be as *orthogonal* as possible. For example, instead of the above definition of *Cancellation* we could also have opted for a weaker variant were a node *v* can also be removed if *Adjacent(v)≠∅*. In that case however, the *Cancellation* would no longer be orthogonal to the *Coarsening*, since all edges adjacent to the removed edge would be removed as well. As another illustration of the orthogonality, our definition of *Refinement* is more restricted than the one in [Steyaert&al96], in which a *Refinement* was also allowed to introduce new nodes. This would imply that a *Refinement* also involves an *Extension*, making the contract types less orthogonal. Another pair of primitive contract types that was present in [Steyaert&al96] was *Concretisation* and *Abstraction*, with the purpose of making an abstract method in a class concrete and vice versa. We have made these productions more general, by introducing a *NodeRetyping* contract type. Moreover, we also extended this idea to edges by introducing an *EdgeRetyping* contract type as well.

Together, the four primitive contract types *Extension*, *Cancellation*, *Refinement* and *Coarsening* form a kind of basis in the mathematical sense of the word, in that every modification of a graph can be expressed in terms of these productions. For example, production *P* in Figure 23 of page 77 can be expressed as a composition of *Extensions* (adding the node *Geo* and its subnodes *perimeter* and *area*), *Cancellations* (removing the subnodes of *Circle* and *Triangle*), *Refinements* (adding the «is-a» edges to *Geo* and adding the *center* edge from *Geo* to *Point*) and *Coarsenings* (removing the *center* edges from *Circle* and *Triangle* to *Point*).

However, in practice it turns out that using only these four primitive contract types is sometimes cumbersome. For example, if we simply want to change the type of a node *v* from υ to ω, we need to do this by first performing a *Cancellation(v,*υ*)* and then reintroducing a new node with the same label but a different type by means of *Extension(v,*ω*)*. The problem even gets worse, since the *Cancellation* can only be applied if *Adjacent(v)=∅*. For this reason, all incoming and outgoing edges in *v* first have to be removed by means of a *Coarsening*, and after the *Extension*, all the edges need to be reintroduced again by means of a *Refinement*. It is obvious that this approach is not very elegant or efficient. For this reason, we have additionally taken the pragmatical decision to introduce *NodeRetyping* and *EdgeRetyping* as primitive contract types.

An advantage of defining the primitive contract types immediately on top of the underlying formalism of graphs and conditional graph rewriting is that we only need to define the primitive contract types once, whereas in [Lucas97] different kinds of *Extensions*, *Refinements*, etc… were distinguished in different situations. For example, a distinction was made between "participant extension" to add an operation to a participant, and "context extension" to add a participant to a collaboration. (A similar distinction could be made for *Refinements*, *Cancellations* and *Coarsenings*.) From a formal point of view, the only difference between both *Extensions* is that they introduce nodes with different types (and consequently also other type constraints). In chapter VI we will show how our primitive contract types can be customised to different domains.

IV 2.2.4 Application Preconditions

From the definitions of the primitive contract types we can immediately deduce the positive and negative application *pre*conditions that need to be valid for the six primitive contract types. These preconditions will allow us to detect some evolution conflicts automatically, and they can also be used to see whether particular sequences of primitive contract types are valid or not.

An example of preconditions in the case of *Refinement(e,v,w,*τ*)* was already given in Figure 20 of section III 3.2.2. More specifically, a morphism *C: L*→*L'* was specified to state that there should be no edge *e* between *v* and *w* before applying the *Refinement*.

To be more concise, we will express L as well as the application preconditions for each primitive contract type $P: L \rightarrow R$ in a single set $PreCond(P)$. Moreover, the preconditions will only be given in textual notation (as defined in sections III 3.5.3 and III 3.5.4). In order to understand the preconditions below, keep in mind that nodes in a graph have a *unique* label (if we disregard nested graphs):

- $\boldsymbol{PreCond(Extension(v,\omega))} = \{ v \notin L \}$ (which implies $Adjacent_L(v)=\emptyset$)
- $\boldsymbol{PreCond(NodeRetyping(v,\omega,\upsilon))} = \{ (v,\omega) \in L \}$ (which implies $(v,\upsilon) \notin L$)
- $\boldsymbol{PreCond(Cancellation(v,\omega))} = \{ (v,\omega) \in L, Adjacent_L(v)=\emptyset \}$
- $\boldsymbol{PreCond(Refinement(e,v,w,\tau))} = \{ v \in L, w \in L, (e,v,w) \notin L \}$
- $\boldsymbol{PreCond(EdgeRetyping(e,v,w,\tau,\phi))} = \{ v \in L, w \in L, (e,v,w,\tau) \in L \}$ (which implies $(e,v,w,\phi) \notin L$)
- $\boldsymbol{PreCond(Coarsening(e,v,w,\tau))} = \{ v \in L, w \in L, (e,v,w,\tau) \in L \}$

In the case of negative conditions, such as $v \notin L$ and $(e,v,w) \notin L$, the type of the node (or edge) is not mentioned. Indeed, to be able to add a node with label v (or edge with label e from v to w) we need to require absence of **all** nodes with label v (or **all** edges with label e from v to w). If the type ω of the node v would be mentioned too, the application precondition would be weaker, and thus not useful. For example, $(v,\omega) \notin L$ would only prohibit the presence of a node v with type ω, while still allowing the presence of a node v with a different type.

Until now, we have only looked at the evolution of graphs without taking type constraints (see section III 2.4 on page 53) on these graphs into account. When type constraints hold between the various nodes and edges of a graph, an extra *implicit* application postcondition holds for the primitive reuse contracts: *a primitive contract type is only applicable to a graph if the resulting graph still satisfies the type constraints*. This condition is guaranteed by requiring that all graphs in the system have the same type graph.

While the preconditions above are sufficient in the domain-independent formalism presented here, additional preconditions might be required when customising the formalism to a specific domain, by defining a type graph and imposing extra type constraints. Obviously, these extra type constraints limit the possible ways in which a given graph can be modified. To cope with this, extra preconditions can be introduced to ensure that the type constraints are still valid after application of the reuse contract. Alternatively, the type constraints could be specified as consistency conditions (Definition 40) on the graph rewriting system that must always be satisfied by each graph.

IV 2.2.5 Application Postconditions

Besides preconditions, each primitive contract type also contains a number of application *post*conditions that need to be present *after* application of each production. For example, after performing an $Extension(v,\omega)$ the node v will be present, and will not contain any incoming or outgoing edges. Again, a set $PostCond(P)$ can be constructed for each primitive contract type $P: L \rightarrow R$. This set expresses R as well as all the application postconditions of P. Again we only use the textual variant.

- $\boldsymbol{PostCond(Extension(v,\omega))} = \{ (v,\omega) \in R, Adjacent_R(v)=\emptyset \}$
- $\boldsymbol{PostCond(NodeRetyping(v,\upsilon,\omega))} = \{ (v,\omega) \in R \}$ (which implies $(v,\upsilon) \notin R$)
- $\boldsymbol{PostCond(Cancellation(v,\omega))} = \{ v \notin R \}$ (which implies $Adjacent_R(v)=\emptyset$)
- $\boldsymbol{PostCond(Refinement(e,v,w,\tau))} = \{ v \in R, w \in R, (e,v,w,\tau) \in R \}$
- $\boldsymbol{PostCond(EdgeRetyping(e,v,w,\phi,\tau))} = \{ v \in R, w \in R, (e,v,w,\tau) \in R \}$ (which implies $(e,v,w,\phi) \notin R$)
- $\boldsymbol{PostCond(Coarsening(e,v,w,\tau))} = \{ v \in R, w \in R, (e,v,w) \notin R \}$

From a theoretical point of view, the postconditions given above are not really essential, since they can be anticipated by an equivalent precondition, as shown in Property 11 of page 69. Nevertheless, for some proofs it will be more convenient to deal with postconditions directly.

IV 2.2.6 Inverse Primitive Contract Types

Convention. From now on, if we use a primitive contract type $(P: L \rightarrow R, ApplCond_P)$, we will always assume that its application conditions specified in $ApplCond_P$ consist of the preconditions $PreCond(P)$ and postconditions $PostCond(P)$ specified in the previous two subsections.

Because the primitive contract types are injective (Property 12), we can *automatically calculate the inverse contract types from the basic ones*. Inverse productions can always be constructed (by inverting all arrows) as long as the production morphism is injective, since the inverse of an injective partial morphism is itself an injective partial morphism. This is exactly what we will do in this subsection.

Using the notion of application preconditions and application postconditions, we can give a formal characterisation of inverse primitive contract types.

If $(P: L \rightarrow R, ApplCond_P)$ is a primitive contract type, then $(Q: R \rightarrow L, ApplCond_Q)$ is its **inverse primitive contract type** (denoted by ***Inverse(P)***) if $PreCond(P) = PostCond(Q)$ and $PostCond(P) = PreCond(Q)$.
If $G \Rightarrow_P H$ is a primitive reuse contract, then $H \Rightarrow_{Inverse(P)} G$ is its **inverse primitive reuse contract**.

Definition 42: Inverse primitive contract types

In other words, Q is the inverse of P if the postconditions of P coincide with the preconditions of Q, and the preconditions of P coincide with the postconditions of Q. This definition also corresponds to the category-theoretical notion of inverse, because if Q is the inverse of P, then $P \circ Q = (id_R: R \rightarrow R, ApplCond_R)$ and $Q \circ P = (id_L: L \rightarrow L, ApplCond_L)$ where $ApplCond_L$ are the preconditions of P, and $ApplCond_R$ are the postconditions of P.

As an immediate result of Definition 42, we can conclude that *Inverse* is an idempotent relation on primitive contract types:

If $(P: L \rightarrow R, ApplCond_P)$ is a primitive contract type, then $Inverse(Inverse(P)) = P$.

Property 13: Idempotence of inverse primitive contract types

When looking carefully at the application *post*conditions and *pre*conditions of all primitive contract type sin the previous subsections, we immediately find the folowing inverse relationships between the six primitive contract types:

Inverse$(Extension(v,\omega)) = Cancellation(v,\omega)$ and vice versa.
Inverse$(Refinement(e,v,w,\tau)) = Coarsening(e,v,w,\tau)$ and vice versa.
Inverse$(NodeRetyping(v,\upsilon,\omega)) = NodeRetyping(v,\omega,\upsilon)$
Inverse$(EdgeRetyping(e,v,w,\tau,\phi)) = EdgeRetyping(e,v,w,\phi,\tau)$

Property 14: Inverse primitive contract types

Proof:

> The proof immediately follows from the observation that
> $PreCond(Extension(v,\omega)) = PostCond(Cancellation(v,\omega))$,
> and similarly for the other primitive contract types.

We can conclude that *Extension* and *Cancellation* are each other's inverse, and similarly for *Refinement* and *Coarsening*. Additionally, *NodeRetyping* and *EdgeRetyping* have themselves as inverse, but with their last two arguments swapped.

IV 2.3 DETECTING INCONSISTENCIES AND CONFLICTS

One of the most important contributions of reuse contracts is that they allow to detect different kinds of interesting evolution conflicts, without needing to resort to sophisticated techniques such as deadlock detection, control-flow analysis, etc. Reuse contracts try to detect as many conflicts as possible by using a limited amount of information only.

In the first section of this chapter we already established the need to distinguish two kinds of conflicts: *applicability conflicts* (corresponding to syntactic inconsistencies) and *evolution conflicts* (corresponding to semantic or behavioural inconsistencies). The next two sections will deal with both kinds of conflicts separately, although a similar approach will be used.

First, a *formal definition* will be given of what a (applicability or evolution) conflict is. Because this definition will be too coarse-grained, we will give a complete *categorisation* of the different kinds of (applicability or evolution) conflicts that can occur. This allows us to be finer-grained, and will make it easier to resolve the conflicts after they have been detected. Note that such a categorisation is only possible because of the fact that we have defined a set of primitive contract types which we are able to express all possible modifications with.

Because conflicts can be detected by comparing pairs of primitive contract types, we can set up *conflict tables* that indicate in which situations a conflict occurs. This is useful, since it will make conflict detection more efficient.

IV . 3 DETECTING APPLICABILITY CONFLICTS

IV 3.1 DEFINITIONS

Applicability conflicts correspond to structural inconsistencies as introduced in the beginning of this chapter. They occur when one graph derivation modifies part of the graph that is required as an *application precondition* for a different graph derivation. Formally, this can be defined in terms of parallel independence (as defined in Definition 30 of page 62).

Let $G \Rightarrow_{P1} G_1$ and $G \Rightarrow_{P2} G_2$ be two primitive reuse contracts. If they are *not* parallelly independent, we say that they lead to an **applicability conflict**.

Definition 43: Applicability conflicts

The following property states that primitive reuse contracts that have *no* applicability conflicts can be serialised. This property is a direct corollary of the local confluence property (Property 4) for conditional derivations. If there is no applicability conflict, the two primitive reuse contracts are parallelly independent. Consequently, the confluence property can be applied, leading to the desired result.

Let $G \Rightarrow_{P1} G_1$ and $G \Rightarrow_{P2} G_2$ be two primitive reuse contracts that do *not* have an applicability conflict. Then $\exists$ unique graph H such that $G \Rightarrow_{P1} G_1 \Rightarrow_{P2} H$ and $G \Rightarrow_{P2} G_2 \Rightarrow_{P1} H$.

Property 15: Serialisation of primitive contract types

In [Lippe&vanOosterom92], a similar approach is taken to see if two transformations P_1 and P_2 of the same graph G can be merged. If transformations P_1 and P_2 commute locally on their initial state G, the final result is a good candidate for the result of the merge.

Because parallel independence is defined in terms of *weak* parallel independence, an applicability conflict occurs either if P_2 is not applicable after P_1, or if P_1 is not applicable after P_2. Moreover, when dealing with conditional productions, this can be detected by a breach of at least one (positive or negative) application precondition (in either P_1 or P_2). This is summarised in the following property.

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

Property 16: Detecting applicability conflicts

Proof:

$G \Rightarrow_{P1} G_1$ and $G \Rightarrow_{P2} G_2$ lead to an applicability conflict
$\Leftrightarrow$ $G \Rightarrow_{P1} G_1$ and $G \Rightarrow_{P2} G_2$ are not parallelly independent
$\Leftrightarrow$ ($G \Rightarrow_{P2} G_2$ is not weakly parallelly independent of $G \Rightarrow_{P1} G_1$) or vice versa
$\Leftrightarrow$ ($PreCond(P_2)$ in G is not preserved by P_1) or vice versa
$\Leftrightarrow$ ($\exists C_2 \in PreCond(P_2)$ such that C_2 is not satisfied in G_1) or vice versa

A direct result of this property is that applicability conflicts can be detected by looking at the pair (P_1, P_2) of primitive contract types, and investigating their interaction. In the next subsection we will give a complete characterisation of all different kinds of applicability conflicts that can be defined in this way, and summarise these results in an *applicability conflict table*.

IV 3.2 CATEGORISATION

IV 3.2.1 Kinds of Applicability Conflicts

By taking into account the fact that we only work with a small number of primitive contract types, we can fine-tune the characterisation of Property 16 by specifying exactly which applicability preconditions are breached in the various situations. Below, all the combinations that lead to an application conflict are discussed using the following template:

AC_i: Name of the applicability conflict

Occurrence. Description of the pair of primitive contract types P_1: $L_1 \rightarrow R_1$ and P_2: $L_2 \rightarrow R_2$ that yield the applicability conflict.
Detection. Description of the application preconditions that are breached in G_1 and/or G_2 when serialising P_1 and P_2.
[optional] Also known as. Name and explanation of the corresponding conflict in [Steyaert&al96] or [Lucas97].

The first two applicability conflicts arise when different primitive contract types add a node with the same label to a graph, or when they both remove a node with the same label from the graph.

AC_1: Duplicate node conflict

Occurrence. P_1=*Extension*(v,ω) and P_2=*Extension*(v,υ) (with possibly $\omega=\upsilon$)
Detection. This conflict is detected by the negative application precondition $\{v \notin L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{v \notin L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.
Also known as. In [Steyaert&al96] this conflict was referred to as an *accidental name collision*. In [Lucas97] it was called a *name conflict*, although a distinction was made between *operation name conflicts* and *participant name conflicts*, depending on whether a participant extension or context extension was used. In our approach, there is no need to make this distinction, since we have only one kind of *Extension* that can be parameterised with different node types.

AC_2: Double cancellation conflict

Occurrence. P_1=*Cancellation*(v,ω) and P_2=*Cancellation*(v,υ)
Detection. This conflict is detected by the positive application precondition $\{v \in L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{v \in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.

The next two conflicts are similar to the dangling edge situation that we already encountered in section III 3.2.2. When an edge is added by one contract type, while the source or target of this edge is removed by a different contract type, we get a dangling edge conflict.

AC_3: Undefined source conflict

Occurrence. P_1=*Refinement*(e,v,w,τ) and P_2=*Cancellation*(v,ω), or vice versa
Detection. This conflict is detected by $\{Adjacent_{L2}(v)=\emptyset\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or by $\{v \in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2. An illustration of this conflict is given in Figure 24.

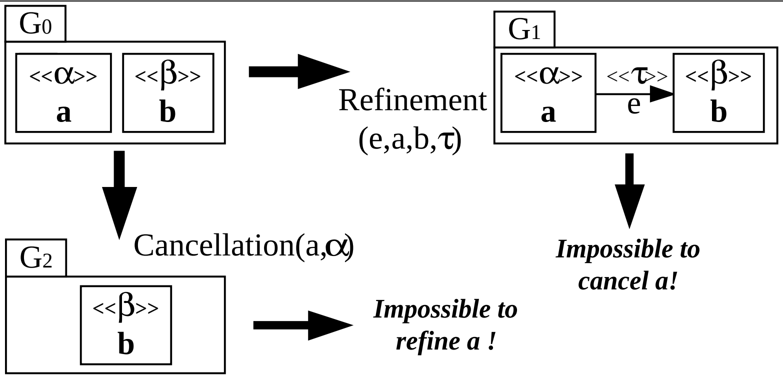


Figure 24: Undefined source applicability conflict

AC_4: Undefined target conflict

Occurrence. P_1=*Refinement(e,v,w,τ)* and P_2=*Cancellation(w,ω)*, or vice versa
Detection. This conflict is the dual of *AC3: Undefined source conflict*. It is detected by $\{Adjacent_{L2}(w)=\emptyset\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or by $\{w\in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.
Also known as. In [Lucas97], this conflict was referred to as a *dangling reference conflict*. It was further subdivided in a *dangling operation conflict* (in the case of a participant refinement and a participant cancellation) and a *dangling participant conflict*. Again, in our approach there is no distinction between both kinds of conflicts except for the type of nodes that are involved.

The following two applicability conflicts are similar to the first two, except that they are defined on edges instead of nodes. When different contract types add an edge with the same label to a graph, or when they both remove an edge with the same label from the graph, we get an applicability conflict.

AC_5: Duplicate edge conflict

Occurrence. P_1=*Refinement(e,v,w,τ)* and P_2=*Refinement(e,v,w,ϕ)*
Detection. This conflict is detected by $\{(e,v,w)\notin L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{(e,v,w)\notin L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.

AC_6: Double coarsening conflict

Occurrence. P_1=*Coarsening(e,v,w,τ)* and P_2=*Coarsening(e,v,w,ϕ)*
Detection. This conflict is detected by $\{(e,v,w)\in L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{(e,v,w)\in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.

Finally, when taking *NodeRetyping* and *EdgeRetyping* into consideration, we get a number of additional conflicts that are very similar to the ones we already encountered above, except that the problems occur at the level of node types instead of node labels.

AC_7: Double node retyping conflict

Occurrence. P_1=*NodeRetyping(v,υ,ω_1)* and P_2=*NodeRetyping(v,υ,ω_2)* (with possibly $\omega_1=\omega_2$)
Detection. This conflict is detected by $\{(v,\upsilon)\in L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{(v,\upsilon)\in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.
Also known as. In [Lucas97], a restricted version of this conflict was referred to as an *annotation conflict*. Annotations *abstract* or *concrete* can be added to each operation, and when two different reusers modify the same annotation, an annotation conflicts arises.

AC_8: Double edge retyping conflict

Occurrence. P_1=*EdgeRetyping(e,v,w,τ,ϕ_1)* and P_2=*EdgeRetyping(e,v,w,τ,ϕ_2)* (with possibly $\phi_1=\phi_2$)
Detection. This conflict is detected by $\{(e,v,w,\tau)\in L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{(e,v,w,\tau)\in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.

AC_9: Undefined node retyping conflict

Occurrence. P_1=*Cancellation(v,ω)* and P_2=*NodeRetyping(v,ω,υ)*
Detection. This conflict is detected by $\{(v,\omega)\in L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{(v,\omega)\in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.

AC_{10}: Undefined edge retyping conflict

Occurrence. P_1=*Coarsening(e,v,w,τ)* and P_2=*EdgeRetyping(e,v,w,τ,ϕ)*
Detection. This conflict is detected by $\{(e,v,w,\tau)\in L_2\} \subseteq PreCond(P_2)$ that is not preserved by P_1 in G_1, or $\{(e,v,w,\tau)\in L_1\} \subseteq PreCond(P_1)$ that is not preserved by P_2 in G_2.

IV 3.2.2 Applicability Conflict Table

Due to the limited set of primitive contract types we have defined, these are the only kinds of applicability conflicts that can occur. They can be summarised in the following symmetric conflict table. Fields containing a √ indicate that there are no applicability conflicts in that specific situation. Shaded fields correspond to one of the applicability conflicts (AC_i) mentioned above. Fields containing an × indicate an impossibility of applying both primitive contract types to the same initial graph. For example, *Extension(v,υ)* and *Cancellation(v,ω)* cannot both be performed on the same graph, since a node *v* cannot be present and absent in a graph at the same time.

	Extend (v,υ)	*Cancel (v,υ)*	*Refine (e,v,w,τ)*	*Refine (e,u,v,τ)*	*Coarsen (e,v,w,τ)*	*Coarsen (e,u,v,τ)*	*Nretype* (v,ω,υ_1)	*ERetype* (e,v,w,τ,ϕ_1)	*ERetype* (e,u,v,τ,ϕ_1)
Extension (v,ω)	AC_1	×	×	×	×	×	×	×	×
Cancellation (v,ω)	×	AC_2	AC_3	AC_4	×	×	AC_9	×	×
Refinement (e,v,w,φ)	×	AC_3	AC_5	√	×	√	√	×	√
Refinement (e,u,v,φ)	×	AC_4	√	AC_5	√	×	√	√	×
Coarsening (e,v,w,φ)	×	×	×	√	AC_6	√	√	AC_{10} if $\phi=\tau$	√
Coarsening (e,u,v,φ)	×	×	√	×	√	AC_6	√	√	AC_{10} if $\phi=\tau$
NodeRetype (v,υ,υ_2)	×	AC_9	√	√	√	√	AC_7	√	√
EdgeRetype (e,v,w,ϕ,ϕ_2)	×	×	×	√	AC_{10} if $\phi=\tau$	√	√	AC_8 if $\phi=\tau$	√
EdgeRetype (e,u,v,ϕ,ϕ_2)	×	×	√	×	√	AC_{10} if $\phi=\tau$	√	√	AC_8 if $\phi=\tau$

Table 1: Applicability conflicts for primitive contract types

IV 3.2.3 Double Check

To ensure that we haven't forgotten any applicability conflicts, we will show that the conflicts identified above are the only ones that are possible. This subsection can be skipped if desired, since it is not necessary in order to understand the rest of the dissertation.

We investigate in which cases $G_0 \Rightarrow_{P1} G_1$ is **not weakly parallelly independent** (Definition 39) of $G_0 \Rightarrow_{P2} G_2$, i.e., when *PreCond*(P_1) is not preserved by P_2 in G_2. Because of Definition 43, all these cases *will coincide with an applicability conflict*.

(a) Let P_2 = *Extension(v,υ)*. The condition $\{v \notin L_1\} \subseteq$ *PreCond*(P_1) is not preserved by P_2 in G_2. Hence P_1 = *Extension(v,ω)*, which requires this precondition, is **not** weakly parallelly independent of P_2. This corresponds to *AC1: Duplicate node conflict.*

(b) Let $P_2 = Cancellation(v,\upsilon)$. The precondition $\{v \in L_1\} \subseteq PreCond(P_1)$ is not preserved in by P_2 in G_2. Hence all the following primitive contract types P_1, requiring this precondition, are **not** weakly parallelly independent of P_2:

- $P_1 = Cancellation(v,\omega)$ *AC2: Double cancellation conflict*
- $P_1 = NodeRetyping(v,\omega,\upsilon)$ *AC9: Undefined node retyping conflict*
- $P_1 = Refinement(e,v,w,\tau)$ *AC3: Undefined source conflict*
- $P_1 = Refinement(e,w,v,\tau)$ *AC4: Undefined target conflict*
- $P_1 = EdgeRetyping(e,v,w,\tau,\phi)$, $P_1 = EdgeRetyping(e,w,v,\tau,\phi)$, $P_1 = Coarsening(e,v,w,\tau)$ and $P_1 = Coarsening(e,w,v,\tau)$: each of these cases is *impossible* since they correspond to an × in Table 1. P_2 requires that $\{Adjacent_{L2}(v)=\emptyset\} \subseteq PreCond(P_2)$, which is incompatible with $\{(e,v,w,\tau) \in L_1\} \subseteq PreCond(P_1)$.

(c) Let $P_2 = Refinement(e,v_1,v_2,\tau)$. The precondition $\{(e,v_1,v_2) \notin L_1\} \subseteq PreCond(P_1)$ is not preserved by P_2 in G_2. Hence $P_1 = Refinement(e,v_1,v_2,\phi)$, which requires this precondition, is **not** weakly parallelly independent of P_2. This corresponds to *AC5: Duplicate edge conflict.*

(d) Let $P_2 = Coarsening(e,v_1,v_2,\tau)$. The precondition $\{(e,v_1,v_2) \in L_1\} \subseteq PreCond(P_1)$ is not preserved by P_2 in G_2. Hence all the following primitive contract types P_1, requiring this precondition, are **not** weakly parallelly independent of P_2:

- $P_1 = Coarsening(e,v_1,v_2,\phi)$ *AC6: Double coarsening conflict*
- $P_1 = EdgeRetyping(e,v_1,v_2,\tau,\phi_2)$ *AC10: Undefined edge retyping conflict*

(e) Let $P_2 = NodeRetyping(v,\upsilon,\upsilon_2)$. The precondition $\{type_{L1}(v)=\upsilon\} \subseteq PreCond(P_1)$ is not preserved by P_2 in G_2. Hence the following primitive contract types P_1, requiring this precondition, are **not** weakly parallelly independent of P_2:

- $P_1 = NodeRetyping(v,\upsilon,\upsilon_1)$ *AC7: Double node retyping conflict*
- $P_1 = Cancellation(v,\upsilon)$ *AC9: Undefined node retyping conflict*

(f) Let $P_2 = EdgeRetyping(e,v_1,v_2,\tau,\tau_2)$. The condition $\{type_{L1}(e,v_1,v_2)=\tau\} \subseteq PreCond(P_1)$ is not preserved by P_2 in G_2. Hence the following primitive contract types P_1, requiring this precondition, are **not** weakly parallelly independent of P_2:

- $P_1 = EdgeRetyping(e,v_1,v_2,\tau,\tau_1)$ *AC8: Double edge retyping conflict*
- $P_1 = Coarsening(e,v_1,v_2,\tau)$ *AC10: Undefined edge retyping conflict*

IV 3.3 Domain-Specific Applicability Conflicts

In Table 1 all the possible applicability conflicts were mentioned. However, by fine-tuning the domain-independent formalism to a specific domain (such as evolution of class diagrams), new *domain-specific applicability conflicts* can be introduced. In section III 2.4 we showed how type graphs and type constraints could be used to attach additional well-formedness rules to domain-specific graphs. Obviously, this has a direct impact on the primitive contract types that can be used. A primitive contract type is only applicable to a domain-specific graph if, besides the usual applicability preconditions, it additionally satisfies the domain-specific type constraints. Each breach of such a type constraint leads to a domain-specific applicability conflict.

Let us illustrate this by means of a concrete graph G, depicted in Figure 25. Suppose that G only contains two nodes $(A,«class») \in G$ and $(B,«class») \in G$ and no edges. Assume that the following edge type constraint holds between the nodes and edges of each domain-specific graph (and, in particular, graph G): "*an «association»-edge is only allowed between «class»-nodes*". Consider a first primitive reuse contract $G \Rightarrow_{P1} G_1$ with contract type $P_1 = NodeRetyping(A,«class»,«interface»)$. It modifies node $(A,«class») \in G$ to $(A,«interface») \in G_1$. Because A has no adjacent edges, the edge type constraint is not broken. Consider a second primitive reuse contract $G \Rightarrow_{P2} G_2$ with contract type $P_2 = Refinement(\varepsilon,A,B,«association»)$ that adds an *«association»*-edge with empty label ε from A to B. Because A and B are *«class»*-nodes in G, the edge type constraints are not broken. However, when serialising both reuse contracts to $G \Rightarrow_{P1} G_1 \Rightarrow_{P2} H$, the resulting graph H does not satisfy the edge type constraints, since it contains an *«assoc»*-edge from an *«interface»*-node to a *«class»*-node. This conflict will be detected as a domain-specific applicability conflict. Indeed, P_2 is not applicable after P_1 because it would break the domain-specific application condition (i.e., the edge type constraint).

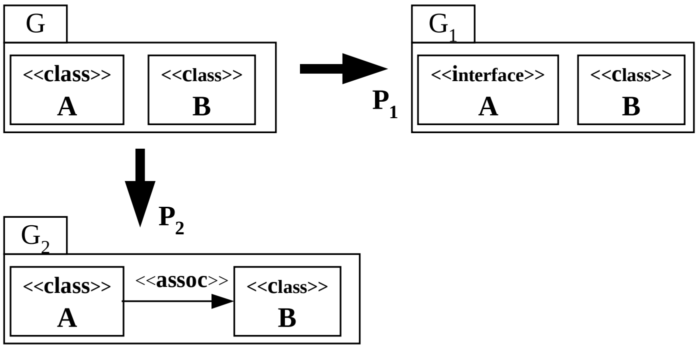


Figure 25: Domain-specific applicability conflict

We will elaborate this topic in chapter VI, where the domain-independent formalism is validated by customising it to different domains.

IV . 4 DETECTING EVOLUTION CONFLICTS

As already mentioned in section IV 1.2, not all inconsistencies between different evolutions of the same software artifact can be detected formally by breaches of application conditions when trying to serialise reuse contracts. So-called *behavioural inconsistencies* arise when independently evolved components yield a result that does not exhibit the expected behaviour: the result does not behave the way developers *assumed* it would. These inconsistencies, which we call *evolution conflicts*, need to be detected by looking for specific graph patterns in the result graph. Alternatively, they can also be detected by means of an evolution conflict table.

In practice, the detection of evolution conflicts is very important, since it allows to analyse the impacts of change, deal with propagation of changes, and deal with proliferation of different versions of the same software artifact [Lucas97]. Moreover, the ability to detect behavioural problems is what distinguishes our approach from most existing merge tools that are only able to detect textual or structural problems.

IV 4.1 INTRODUCTORY EXAMPLE

As a concrete example, reconsider the *Circle* object of Figure 23 on page 77, with attributes *radius*, *perimeter* and *area* as subnodes. Again, keep in mind that nesting is only used to make the example more readable, and will be dealt with more elaborately in chapter V.

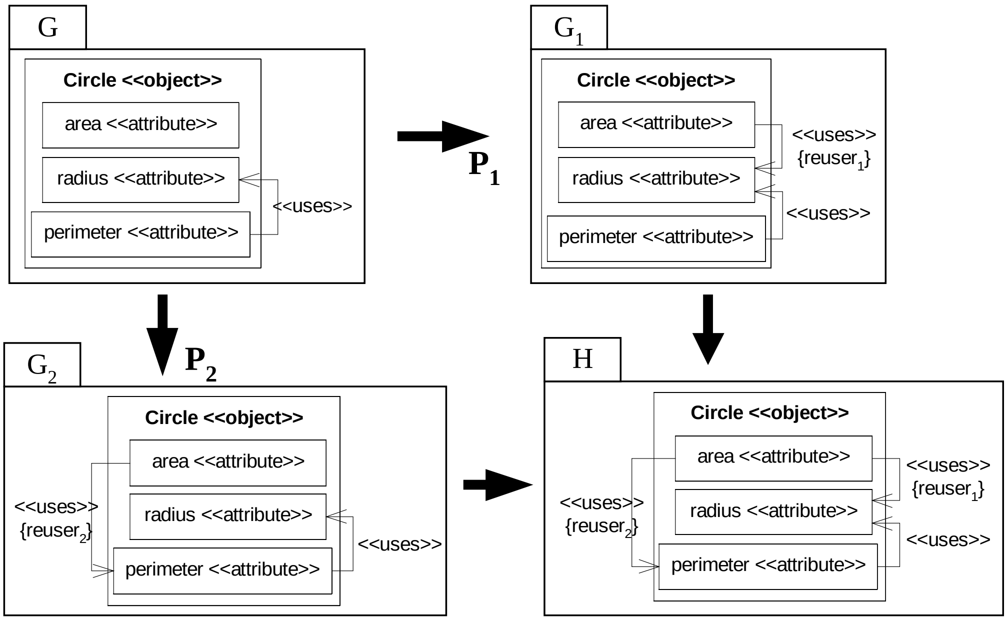


Figure 26: Double reachability conflict

Figure 26 shows the *Circle* object and two parallel modifications. Because the *perimeter* of an object can be automatically derived from the *radius* (it is a so-called derived attribute), an edge with empty label ε and type *«uses»* is placed between *perimeter* and *radius*. Now suppose that this basic design, specified in *G*, is modified by different evolvers in different ways. The first evolution is specified by a reuse contract $G \Rightarrow_{P1} G_1$ with contract type P_1 = *Refinement(ε,area,radius,«uses»)*. This modification adds a *«uses»*-edge from *area* to *radius*, to indicate that the *area* can be automatically derived from the *radius*. The second evolver takes an alternative approach, by deriving *area* from *perimeter* (the *area* can be obtained by integrating the *perimeter*). This is achieved by a reuse contract $G \Rightarrow_{P2} G_2$ with contract type P_2 = *Refinement(ε,area,perimeter,«uses»)*. Because there are no applicability conflicts, both reuse contracts can be serialised, and lead to a unique result graph *H*. In this result graph, however, *radius* suddenly can be reached from *area* via two different paths: a direct edge, and a path of length two via *perimeter*. Because this ambiguity might not have been intended by one or both evolvers, we say that an **evolution conflict** (more specifically, a **double reachability conflict**) has occurred. To resolve the conflict, a human decision needs to be made to decide which of both paths is more appropriate.

IV 4.2 Discussion

Evolution conflicts typically occur during collaborative software development when *different* persons *independently* evolve the same software artifact. If they make this modification for the same reason (e.g., the same bug fix, or adding the same functionality), an applicability conflict is very likely to occur because the same parts of the software will be modified twice. If both modifications are made for different purposes, evolution conflicts may be introduced due to unforeseen interactions between these modifications.

Note that both modifications do not necessarily have to be made by different persons. A software engineer who wants to make modifications to a piece of software might decide to make various modifications in parallel, and merge them afterwards. During this merge, evolution conflicts might again show up, although these evolution conflicts are less likely than in the case of different developers. Nevertheless, if a significant amount of time has passed between different modifications made to the same software artifact by the same developer, the developer might have forgotten the specific details of his previous modification, again increasing the likelihood of an evolution conflict.

From this discussion we can conclude that there are *various degrees of likelihood* of an evolution conflict occurring, depending on who makes the modifications, why the modifications are made, and when they are made. Formally, this can be dealt with by annotating or *tagging each modification* with the name, intention (e.g., bug fix, or name of the requirement that is being implemented) and timestamp of the software developer that is making the change. (Many other tags are possible.) Based on these tags we can then decide whether or not it is necessary to engage our conflict detection algorithm.

In order to add these so-called *modification tags*, we do not need to extend our formalism, since we can simply make use of constraints that can be attached to individual nodes and edges. In Figure 26 the constraints *{modification-tag =" reuser$_1$"}* and *{modification-tag =" reuser$_2$"}* (abbreviated to *{reuser$_1$}* and *{reuser$_2$}*) were added to the newly introduced edges to show that different persons have made both modifications. In [DeHondt98] it is explained in detail how a tagging tool can be (and *is*) used in practice.

It is not the scope of this dissertation to present an extensive treatment of all aspects involved in this tagging, and how this can influence the likelihood of an evolution conflict. We will just take the basic approach of making a distinction between absence of an evolution conflict if both interacting modifications have the same modification tag, and the presence of an evolution conflict if both modifications have a different modification tag. When implementing the formalism in a tool, it is clear that this basic approach should be further refined.

IV 4.3 Conditional Approaches to Conflict Detection

Just like *applicability* conflicts could be detected by making use of application conditions attached to productions, it is also possible to detect *anticipated evolution conflicts* by means of these same application conditions. To achieve this, application conditions must now be attached to the graph that evolves (to represent *evolution assumptions*), or to the graph rewriting system (to specify *evolution invariants*). Both approaches will be explained below. The evolution conflicts that can be detected in this way are “anticipated” because the application conditions that have been added allow us to specify *in advance* which kinds of modifications are prohibited. However, because the basic idea of reuse contracts is to detect *unanticipated* evolution conflicts rather than anticipated ones, we will not discuss these approaches in full detail.

IV 4.3.1 Specifying Evolution Assumptions

In [Lehman98], several problems concerning software evolution are discussed. As one of the recommendations to solve these problems, Lehman proposed to capture and record all *implicit assumptions* in the software. This must be done “in a structured fashion to simplify the inspection and identification of any assumptions that may have become of questionable validity”. The importance of these assumptions for detecting evolution conflicts is clearly stated: “Proposed changes to a software system must be examined in relation to the existing bounds and assumption set to avoid incompatibility or other undesirable side effects”.

Once the important assumptions have been identified, our formal model allows us to make these assumptions explicit by attaching extra application conditions to the graph that is to be modified. In this way, anticipated evolution conflicts can be detected in a way similar to the applicability conflicts of section IV . 3 . Although this approach ensures that more conflicts will be detected than only the basic applicability conflicts, the software developer might not always have a clear idea about the implicit assumptions that must hold in a particular software component. Moreover, even if the assumptions are clear, there might be so many assumptions that it is unfeasible to make them all explicit.

Formally, many **assumptions** can be made explicit by using the notion of *conditional graphs* (as defined in Definition 37). To each graph G with match $m: L \rightarrow G$, a set $EvolCond(L)$ is attached representing the assumptions that should not be breached during evolution. Then, anticipated evolution conflicts as well as applicability conflicts can both be expressed by making use of application conditions (as defined in Definition 36 of page 66). The only difference is that *applicability conflicts* will be detected by breaches of application preconditions that are part of the primitive contract types (formally represented as *conditional productions*), while *anticipated evolution conflicts* will be detected by breaches of application conditions that are directly attached to a graph (using *conditional graphs*).

Let $(G_0, m_1: L_1 \rightarrow G_0, EvolCond_1(L_1))$ and $(G_0, m_2: L_2 \rightarrow G_0, EvolCond_2(L_2))$ be two conditional graphs and $(P_1: L_1 \rightarrow R_1, ApplCond_1(L_1))$ and $(P_2: L_2 \rightarrow R_2, ApplCond_2(L_2))$ two primitive contract types such that the primitive reuse contracts $G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P2,m2} G_2$ are parallelly independent.

$G_0 \Rightarrow_{P1,m1} G_1$ and $G_0 \Rightarrow_{P2,m2} G_2$ lead to an **anticipated evolution conflict** if and only if $\exists\, C_2 \in EvolCond_2(L_2)$ that is not satisfied in G_1, or $\exists\, C_1 \in EvolCond_1(L_1)$ that is not satisfied in G_2.

Definition 44: Detecting anticipated evolution conflicts

Observe the similarity between the above definition and Property 16. Applicability conflicts and anticipated evolution conflicts are detected in the same way, but with other application conditions. According to the definition above, an anticipated evolution conflict only arises when $EvolCond_1(L_1)$ is not preserved by P_2 or when $EvolCond_2(L_2)$ is not preserved by P_1. Obviously, P_1 itself is allowed to make some changes to the evolution conditions imposed by $EvolCond_1(L_1)$, since the implicit assumptions will evolve when G_0 evolves. As a result it is possible (and even likely) that $EvolCond_1(L_1)$ is not satisfied in G_1, but this is of course not considered to be an evolution conflict.

IV 4.3.2 Specifying Evolution Invariants

As an alternative to specifying a set of assumptions as evolution conditions on a graph each time we perform an evolution step on this graph, we can also specify **evolution invariants**. These invariants are assumptions that must be maintained during the entire evolution process. In other words, these assumptions should never be invalidated. They can be considered as architectural principles that should be preserved throughout the (evolutionary) life time of the software system. From a formal point of view, they can be dealt with by attaching *consistency conditions* (see Definition 40 on page 70) to the graph rewriting system instead of to the graph that is evolving. All the initial graphs in the graph rewriting system need to satisfy these consistency conditions, and after applying productions, the conditions still need to be satisfied.

Intuitively, evolution invariants can also be seen as extra well-formedness constraints that must be satisfied by each graph in the graph rewriting system at all times. A benefit of this approach is that we can specify in advance which constraints in a graph should certainly not be broken upon evolution. These constraints can be considered as **architectural invariants**, because they are important in the sense that, if one of them becomes invalidated, the entire software architecture might collapse. When we draw the analogy with architecture of buildings, these constraints would correspond to supporting beams or supporting walls of a building. If one of these elements is removed from the building, the entire building can collapse. Observe that the same idea of ensuring that evolution invariants are maintained by a software system has also been proposed by Marvin Minsky, who uses *law-governed architectures* and *law-governed interactions* to preserve these invariants. Another related approach is [LeMétayer98], where graph grammars have been used to describe software architecture styles. While software architectures are represented as graphs (with nodes representing components and edges representing their interconnection), an architecture style is defined as a class of software architectures specified by a graph grammar.

Note that the *type graph* can be considered as a specific kind of evolution invariant. Indeed, all labelled typed graphs have to satisfy the same type graph during evolution.

An important benefit of *evolution invariants* (as well as the *evolution assumptions* of the previous section) is that they allow us to express *negative information* in the contract clauses of a reuse contract. Until now, this was not possible with reuse contracts, and in several experiments this has shown to be a real restriction. To express negative information, if suffices to use negative application conditions as invariants (or as assumptions).

The use of invariants, preconditions and postconditions on graphs bears many similarities to the *"design by contract"* approach of [Meyer92]. Invariants, preconditions and postconditions are introduced there at the level of methods and classes in the object-oriented programming language Eiffel. In our approach, these ideas are generalised to arbitrary components (not necessarily restricted to the implementation level), as long as they can be expressed by means of labelled typed graphs.

IV 4.4 Reuse Contract Approach to Conflict Detection

IV 4.4.1 Unanticipated Evolution

Although the approaches towards evolution that are mentioned in the previous section are certainly useful, they have the important limitation that they can only be used to detect *anticipated* evolution conflicts: they detect problems involving *evolution assumptions* and *evolution invariants* that were specified in advance. Because it is practically infeasible to identify all necessary assumptions and invariants that must hold during evolution, there will always be some conflicts that will not be detected by these approaches.

To cope with this situation, reuse contracts [Steyaert&al96, Lucas97] take the alternative approach of detecting *unanticipated* evolution conflicts. Additionally, the kind of conflicts that we try to detect are *behavioural* inconsistencies when merging parallel evolutions of the same software artifact. Obviously, because any nontrivial property concerning the behaviour of a program is inherently undecidable [Rice53], it is impossible to develop an algorithm that decides whether a merge of two parallel evolutions leads to a behavioural inconsistency. The best we can do is provide a *safe* approximation that warns the user when a merge is *potentially* semantically incorrect. This is a *conservative* approach, and can sometimes lead to a large number of unnecessary conflict warnings. Therefore, formal techniques are needed that allow us to reduce the evolution conflicts to a manageable number.

IV 4.4.2 Example

Let us now try to translate this idea in terms of conditional graph rewriting and category theory. Therefore, we reconsider the example of Figure 26, but now expressed directly in terms of objects and morphisms in the category ***LTGraphL*** (i.e., labelled typed graphs and label-preserving morphisms).

The reuse contract $G \Rightarrow_{P1,\, m1} G_1$ actually consists of two morphisms $(P_1{:}\ L_1 \rightarrow R_1,\ m_1{:}\ L_1 \rightarrow G)$ together with their pushout $(G_1,\ P_1{*}{:}\ G \rightarrow G_1,\ m_1{*}{:}\ R_1 \rightarrow G_1)$. Similarly, the reuse contract $G \Rightarrow_{P2,\, m2} G_2$ consists of two morphisms $(P_2{:}\ L_2 \rightarrow R_2,\ m_2{:}\ L_2 \rightarrow G)$ together with their pushout $(G_2,\ P_2{*}{:}\ G \rightarrow G_2,\ m_2{*}{:}\ R_2 \rightarrow G_2)$. Because the two reuse contracts are parallelly independent, they can be serialised according to the local confluence property, and lead to a result graph H that incorporates the changes of both modifications. Formally, this is achieved by calculating the *pushout* of $P_1{*}{:}\ G \rightarrow G_1$ and $P_2{*}{:}\ G \rightarrow G_2$, and is represented by dashed arrows in the lower right of Figure 27.

In order to know whether there will be a potential evolution conflict, we need to find out if both reuse contracts perform modifications which might have possible side effects on each other. This is the case if a node or edge plays a role in both modifications at the same time, i.e., if there is a common part in both modifications. In the example of Figure 27 this is indeed the case, since the node *(area, «attribute»)* plays a role in P_1 as well as in P_2. Indeed, it occurs on the left-hand side of both productions. Formally, to find all nodes and edges that play a role in two productions at the same time, we can calculate the *pullback* of $m_1{:}\ L_1 \rightarrow G$ and $m_2{:}\ L_2 \rightarrow G$, represented by dashed arrows in the upper left of Figure 27. In this specific example, *area* is the only node that leads to a potentially undesired interaction. The particular behavioural problem is called a "double reachability conflict", and has already been explained in more detail in section IV 4.1.

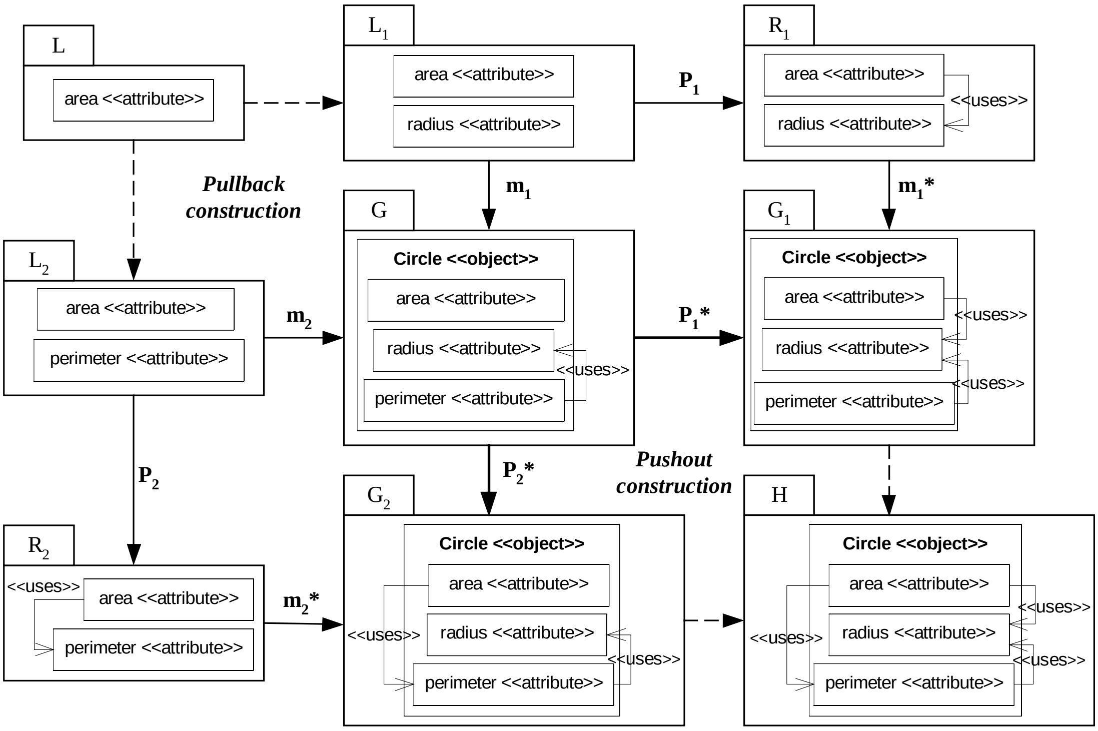


Figure 27: Formal detection of potential evolution conflicts

IV 4.4.3 Definition

The above discussion allows us to give a formal definition of a potential evolution conflict:

> Let $G \Rightarrow_{P1,\, m1} G_1$ and $G \Rightarrow_{P2,\, m2} G_2$ be two parallelly independent primitive reuse contracts. They lead to a **potential evolution conflict** if the pullback L of m_1: $L_1 \rightarrow G$ and m_2: $L_2 \rightarrow G$ is not empty.

Definition 45: Potential evolution conflicts

The requirement that the reuse contracts should be parallelly independent is needed to ensure that they do not lead to an applicability conflict.

Obviously, this definition gives a very rough approximation of when an evolution conflict occurs. Therefore, we will take the same approach as with applicability conflicts to try and find a finer-grained characterisation, based on the fact that we only have a limited set of primitive contract types. For each pair of primitive contract types we can give a characterisation of when they lead to a potentially undesired interaction, as well as a more detailed description and intuitive explanation of the particular evolution conflict. As for applicability conflicts, all different kinds of evolution conflicts can be put in a conflict table, which allows us to detect them more efficiently by performing a table-lookup.

IV 4.4.4 Graph Patterns

In this subsection we present an alternative way for detecting evolution conflicts. This alternative approach, which is more elegant and useful for theoretical purposes, goes as follows. For each pair of parallelly independent modifications of the same initial graph, detecting if an evolution conflict occurs between them corresponds to finding a particular *graph pattern* in the result of serialising both modifications.

Figure 28 presents an example of such a graph pattern that can be used to detect the evolution conflict of Figure 27. This so-called *double reachability conflict* occurs when the independent evolution steps P_1 and P_2 give rise to two different paths of «uses»-edges between «attribute»-nodes. Moreover, the different paths must have been created by different persons (in this case, *{reuser$_1$}* and *{reuser$_2$}*). If both paths have length one (i.e., they are direct edges between the nodes), they can be detected by looking for the graph pattern of Figure 28 in the result graph H of Figure 27. If one or both paths have length two (as is the case in the considered example), we can only detect the conflict by looking for the graph pattern in the second-order closure H^2, as defined in Definition 10 of page 46. In general, if we want to be able to detect the conflict with paths of arbitrary length, we need to find the graph pattern in the transitive closure graph H^+. More about this will be said in subsection IV 4.5.3.

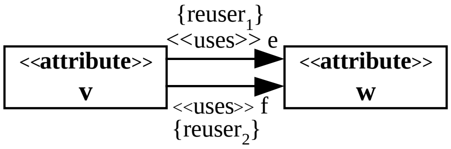


Figure 28: Double reachability graph pattern

If *EvolConf* represents the set of all possible graph patterns that lead to a potential evolution conflict, then looking for a graph pattern $C \in EvolConf$ in the result graph H can be achieved by finding an injective match $m: C \rightarrow H$. This finer-grained characterisation of potential evolution conflicts is given below.

Let *EvolConf* be the set of all possible **evolution conflict patterns**. Let $G \Rightarrow_{P1,m1} G_1$ and $G \Rightarrow_{P2,m2} G_2$ be two parallelly independent primitive reuse contracts. Their sequentialisation $G \Rightarrow_{P1,m1} G_1 \Rightarrow_{P2,n2} H$ (with $n_2 = P_1{*} \circ m_2$) leads to a **potential evolution conflict** if $\exists C \in EvolConf$: $\exists$ injective match $m: C \rightarrow H$.

Definition 46: Finer-grained characterisation of potential evolution conflicts

The match m is not required to be label-preserving. In other words, the node labels v and w and edge labels e and f of Figure 28 can be mapped on any node label and edge label in H, as long as they are mapped on different nodes and different edges (because of injectivity). Additionally, the match should ensure that e and f are introduced by different reusers, by checking that the modification tags are different.

Definition 46 should be seen as a way to refine the general characterisation of Definition 45 to make a distinction between different kinds of evolution conflicts. More specifically, each conflict pattern will give rise to a different evolution conflict. In general, detection of conflict patterns in a graph is very inefficient, because finding a certain subgraph in an arbitrary graph is an NP-complete problem. Fortunately, in our case the search problem can be reduced significantly, because we only have to look at graph patterns that involve those nodes or edges that are present in the pullback of Definition 45. Hence, if the pullback is not too large (which signifies the presence of a large number of potential evolution conflicts), the graph patterns can be detected relatively fast.

An important difference with the approach of detecting conflicts by comparing primitive contract types is that graph patterns are only checked in the result graph H **after** having serialised both primitive reuse contracts.[1]

IV 4.4.5 Categorisation of Evolution Conflicts

Below, we will discuss all possible kinds of evolution conflicts that can occur when two primitive reuse contracts modify the same graph. Each of these conflicts can be detected by first checking if the pullback is nonempty (Definition 45), and then detecting a particular graph pattern in the result graph (Definition 46). Like with applicability conflicts, evolution conflicts will only arise for particular combinations of primitive contract types, which will allows us to set up an *evolution conflict table* as well. All evolution conflicts will be discussed using the following template:

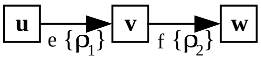
[1] The attentive reader may have observed that this approach cannot be used directly when the reuse contract types P_1 and P_2 *remove* items from the graph (by means of *Cancellation* or *Coarsening*). In a later subsection we will show how the approach can be extended to deal with those modifications as well.

EC_i: Name of the potential evolution conflict

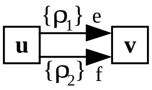
Conflict pattern. Description of the evolution conflict pattern that identifies the potential evolution conflict.
Occurrence. Description of the node in the pullback that gives rise to a potentially undesired interaction, as well as a description of the pair of primitive contract types that lead to the evolution conflict.
Explanation. Informal description of the evolution conflict, and why it is considered as a potential problem.
[optional] Remarks. Any additional comments concerning the specific evolution conflict.
[optional] Also known as. Whenever relevant, we mention how the evolution conflict is related to the ones discussed in [Lucas97]. In chapter VI we will discuss this in more detail by showing that our approach is a generalisation (and formalisation) of [Lucas97].

All evolution conflicts will be expressed in terms of graph patterns. Existence of such a pattern in the result graph H corresponds to the occurrence of an evolution conflict. In most cases, the type of the nodes and edges will be omitted in the graph pattern, indicating that the node types are irrelevant for detecting the conflict. Also, in all the evolution conflicts below, we assume that we start from an initial graph G_0 in which the modification tags of all the nodes and edges are empty (i.e., *{}*).[2] Modifications performed by primitive contract type P_1 will have the side-effect of adding the tag $\{\rho_1\}$ to all modified nodes and edges. Modifications performed by primitive contract type P_2 will have the side-effect of adding the tag $\{\rho_2\}$ to all modified nodes and edges. Consequently, after merging both parallel productions, we get a result graph H in which all modified or newly introduced nodes and edges are tagged with either $\{\rho_1\}$ or $\{\rho_2\}$, while all nodes or edges that remain untouched are tagged with *{}*. For all the evolution conflicts, we will assume that $\rho_1 \neq \rho_2$, since otherwise both modifications have been made by the same person for the same purpose, and in those cases an evolution conflict is less likely.

EC_1: Reachability conflict

Conflict pattern. A reachability conflict is detected by .
Occurrence. It occurs when v plays a role in both reuse contracts. More specifically, the conflict can only occur when reuser ρ_1 performs a $Refinement(e,u,v,\tau)$ or $EdgeRetyping(e,u,v,\tau_1,\tau)$ with target node v while reuser ρ_2 performs a $Refinement(f,v,w,\phi)$ or $EdgeRetyping(f,v,w,\phi_2,\phi)$ with source node v.
Explanation. This conflict arises whenever the result graph contains three different nodes (with arbitrary node labels and node types) connected by edges that have been introduced by different modifiers (since they correspond to different tags). It is a conflict when reuser ρ_1 introduces an edge e between u and v, with the implicit assumption that v should not reach w. When reuser ρ_2 introduces a different edge f between v and w, this assumption becomes broken.
Remarks. Note that, in the graph pattern above, it is important that u and w are different nodes (which is ensured by using injective matches). If u and w were the same node, we would have an *EC5: Cycle introduction conflict*, which is explained later.
Also known as. In [Lucas97], the term *operation capture* is used to deal with reachability conflicts, in the situation where two operation invocations are introduced by means of a participant refinement.

EC_2: Double reachability conflict

Conflict pattern. A double reachability conflict is detected by .
Occurrence. It occurs when u and v play a role in both reuse contracts. More specifically, the conflict can only occur when reuser ρ_1 performs a $Refinement(e,u,v,\tau)$ or $EdgeRetyping(e,u,v,\tau_1,\tau)$ while reuser ρ_2 performs a $Refinement(f,u,v,\phi)$ or

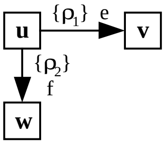
[2] In practice, this assumption is too restrictive, but we will see in subsection IV 4.6.2 how this problem can be resolved.

EdgeRetyping(f,u,v,ϕ_2,ϕ) with the same source and target nodes. If *e=f* this is detected by application condition *AC5: Duplicate edge conflict.*

Explanation. This conflict arises whenever the result graph contains two different nodes (with arbitrary node labels and node types) connected by two different edges with the same direction, that have been introduced by different reusers (since they correspond to different tags). This conflict has already been illustrated in Figure 26 on page 91.

Remarks. Actually, this evolution conflict can be seen as the combination of the next two conflicts.

EC_3: Double source conflict

Conflict pattern. A double source conflict is detected by 

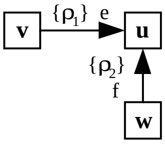
.

Occurrence. It occurs when *u* plays a role in both reuse contracts. More specifically, the conflict can only occur when reuser ρ_1 performs a *Refinement(e,u,v,τ)* or *EdgeRetyping(e,u,v,τ_1,τ)* while reuser ρ_2 performs a *Refinement(e,u,w,ϕ)* or *EdgeRetyping(e,u,w,ϕ_2,ϕ)* with the same source node.

Explanation. In some sense, this conflict can be considered as a variant of *EC2: Double reachability conflict*. It occurs when more than one edge with the same source node is introduced by different reusers, as long as they lead to a different target node.

Also known as. In [Lucas97], this conflict is called an *operation invocation conflict* when two different reusers perform a participant refinement of the same operation, i.e., the specialisation clause of an operation (which determines which operation invocations this operation performs) is augmented by two different persons. The same conflict can also occur with two context refinements of the same participant, but then it is called an *acquaintance relationship conflict*.

EC_4: Double target conflict

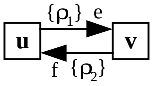
Conflict pattern. A double target conflict is detected by 

.

Occurrence. It occurs when *u* plays a role in both reuse contracts. More specifically, the conflict can only occur when reuser ρ_1 performs a *Refinement(e,v,u,τ)* or *EdgeRetyping(e,v,u,τ_1,τ)* while reuser ρ_2 performs a *Refinement(e,w,u,ϕ)* or *EdgeRetyping(e,w,u,ϕ_2,ϕ)* with the same target node.

Explanation. This conflict is the dual of *EC3: Double source conflict*. In some sense it can be considered as a variant of *EC2: Double reachability conflict*, but with different source nodes. It occurs when two different reusers add a different edge with the same target node.

EC_5: Cycle introduction conflict

Conflict pattern. A cycle introduction conflict is detected by 

.

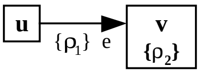
Occurrence. It occurs when nodes *u* and *v* play a role in both reuse contracts. More specifically, the conflict can only occur when reuser ρ_1 performs a *Refinement(e,u,v,τ)* while reuser ρ_2 performs a *Refinement(e,v,u,τ)* with source and target nodes swapped.

Explanation. This conflict arises whenever the result graph contains two different nodes (with arbitrary node labels and node types) connected by two different edges *in the opposite direction* that have been introduced by different reusers (since they correspond to different tags).

Also known as. This conflict is referred to as *unanticipated recursion* in [Lucas97].

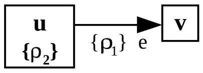
All the evolution conflicts discussed so far were caused by two different *Refinements* (or *EdgeRetypings*) introduced by different reusers. There are however some other evolution conflicts that can occur as well:

EC_6: *Inconsistent target conflict*

Evolution condition. An inconsistent target conflict is detected by 

.
Occurrence. It occurs when node *v* plays a role in both reuse contracts. More specifically, the conflict can only occur when reuser ρ_1 performs a *Refinement(e,u,v,τ)* or *EdgeRetyping(e,u,v,τ_1,τ)* while reuser ρ_2 performs a *NodeRetyping(v,υ,ω).*
Explanation. This conflict arises whenever an edge is added to a node by one reuse contract, while this target node is modified in some way by a different reuse contract. Indeed, the reuser that added the edge might have assumed particular properties about the target component that might be invalidated by the reuse contract of the second reuser.

EC_7: *Inconsistent source conflict*

Conflict pattern. An inconsistent source conflict is detected by 

.
Occurrence. It occurs when node *u* plays a role in both reuse contracts. More specifically, the conflict can only occur when reuser ρ_1 performs a *Refinement(e,u,v,τ)* or *EdgeRetyping(e,u,v,τ_1,τ)* while reuser ρ_2 performs a *NodeRetyping(u,υ,ω).*
Explanation. This conflict is the opposite of *EC6: Inconsistent target conflict*. It occurs when one reuse contract modifies a particular node *u*, while a different reuse contract introduces a new edge with *u* as source node. Again, this is a potential conflict since the reuser that introduces the edge is not aware of the changes to the source node which might break particular assumptions required by the edge.

IV 4.5 FINE-TUNING THE EVOLUTION CONFLICTS

Note that the potential evolution conflicts mentioned above are still too general, in the sense that they detect too many unnecessary conflict warnings in practical situations. Therefore we need some mechanisms to reduce the number of unnecessary warnings based on domain-specific knowledge. This will be discussed in subsection IV 4.5.1.

On the other hand, the presented evolution conflicts are still too restrictive for the following reasons:

- The described potential evolution conflicts can only detect problems based on the *presence* of edges between nodes. For this reason, we will not be able to detect evolution conflicts if one or both evolvers performs a *Coarsening*. In subsection IV 4.5.2, we shall generalise the approach to deal with evolution conflicts based on *absence* of edges between nodes.
- The evolution conflicts are only useful for dealing with conflicts based on *direct* edges (i.e., paths of length one) between nodes. This is however too restrictive in practice. For example, the current version of *EC5: Cycle introduction conflict* can only be used to detect cycles of length two. Similar restrictions hold for the other potential evolution conflicts. For example, the *EC2: Double reachability conflict* of Figure 26 can only be detected if we also take paths of length two into consideration. In subsection IV 4.5.3 we will show how these problems can be overcome by making use of the *transitive closure*.

IV 4.5.1 Domain-specific Evolution Conflicts

When customising the domain-independent formalism to specific application domains, the evolution conflicts can be fine-tuned by restricting the occurrence of an evolution conflict based on the domain-specific types of the nodes and edges that are present in the conflict pattern. This allows us to reduce the number of evolution conflicts that will actually be detected in a specific domain. For example, one possible restriction could be that the *EC1: Reachability conflict* (as well as other evolution conflicts) should be detected only when the two edges that are involved in the graph pattern have *the same edge type*. An even further restriction could be that this edge type must be *«invocation»* while the types of the nodes must be *«operation»*. (This is the case in [Lucas97], where the corresponding conflict is referred to as *operation capture*.)

From the above discussion we can conclude that the formal conflict detection approach explained in this chapter assumes a “worst case” scenario. Due to the absence of domain-specific knowledge, many conflict warnings will be generated that do not correspond to real conflicts when customising the formalism to a specific domain. Hence, domain-specific information allows us to ignore a lot of potential evolution conflicts that are generated by the conflict detection algorithm.

Those evolution conflicts that remain could be sorted in order of importance, e.g., based on the edge types that are involved. For some edge types, the evolution conflicts will be more severe than for others. Other more sophisticated techniques could be invented to determine the importance of an evolution conflict.

Another way to reduce the domain-specific evolution conflicts is by introducing domain-specific type constraints. In section IV 3.3 we explained how extra type constraints gave rise to domain-specific applicability conflicts. As a result, all evolution conflicts that coincide with these domain-specific applicability conflicts do not have to be detected any longer. Let us explain this by means of an example. Suppose that the domain-specific type constraint is: “*an «object»-node cannot be the source of more than one «instance»-edge*” (which is a so-called *multiplicity constraint*). In the domain-independent formalism, this could be detected by looking for an *EC3: Double source conflict*. In the domain-specific customisation, this evolution conflict will always coincide with a breach of the type constraint, so the evolution conflict will no longer occur. Instead, a domain-specific applicability conflict arises.

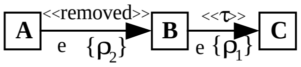
It remains to be seen in practice which other useful restrictions can be made to reduce the detected evolution conflicts to a manageable number.

IV 4.5.2 Conflicts with Cancellation or Coarsening

A problem with the detection of graph patterns is that the approach is useless when nodes or edges are *removed* during a graph modification by means of a *Cancellation* or *Coarsening*. In those cases it is impossible to know which reuser performed the modification, since the nodes or edges are no longer present in the result graph. This is the reason why all the evolution conflicts considered above were caused by *Refinements* or *EdgeRetypings*. However, similar conflicts also occur when *Coarsenings* are being made. In order to be able to detect these conflicts by looking at the result graph only, we need to make a small but important modification to our formalism.

In order to know if a node or edge has been removed by a particular reuser, we should not really remove this node or edge in the result graph, but simply give it a new type *«removed»*. If we find a node with modification tag $\{\rho_1\}$ and type *«removed»* in the result graph, we know that it has been removed by reuser ρ_1 using a *Cancellation*. Similar, if we find an edge with modification tag $\{\rho_1\}$ and type *«removed»*, we know that it has been removed by reuser ρ_1 using a *Coarsening*.

While this is only a simple change to the formalism (the introduction of a new node type and edge type *«removed»* with special semantics, as well as a minor revision of the primitive contract types), it has an important advantage. All the evolution conflicts discussed above will be immediately applicable for *Coarsenings* as well, by using the specific edge type *«removed»* for the edges involved in the graph pattern. As a result, the evolution conflicts will also occur if a *Coarsening* is performed instead of a *Refinement* or *EdgeRetyping*.

As an example of such a conflict, suppose that we have a graph containing three nodes with labels *A*, *B* and *C*, as well as an edge with label *e* and type τ between *A* and *B*. One reuser adds an edge with label *e* and type τ from *B* to *C*, with the explicit intent of reaching *C* indirectly from *A*. A second reuser independently removes the edge from *A* to *B*, unaware of the intentions of the first reuser. Then we clearly have a conflict, which is detected by means of the graph pattern 

. This graph pattern is an instance of *EC1: Reachability conflict*. In the same way, all other conflicts with *Coarsening* or *Cancellation* are instances of existing evolution conflicts.

It should be noted that, in practice, the evolution conflicts that arise because of a *Refinement* or *EdgeRetyping* are usually more severe than the conflicts that involve a *Coarsening*. Therefore, a conflict detection tool should allow to ignore the latter conflicts if necessary, in order to avoid generating too many conflict warnings at the same time.

IV 4.5.3 Transitive Closure

In some cases we need to make use of the transitive closure to find out whether there is a conflict. Conflicts that can only be detected by taking the transitive closure into account are called **transitive conflicts**. For example, with *EC5: Cycle introduction conflict*, we can only detect cycles of length two. With a transitive closure, we are able to detect cycles of arbitrary length. Actually, for all the evolution conflicts mentioned in section IV 4.4, there is a corresponding transitive evolution conflict. These conflicts can be detected in precisely the same way as their simple variants, except that we need to work with the transitive closure graph.

We can distinguish different gradations of conflicts. A **first-order conflict** is an evolution conflict that can be detected by looking at direct edges in the result graph *H*. **Second-order conflicts** are conflicts that can only be detected by looking at indirect edges of order 2, corresponding to paths of length 2 or less in the original graph. In a similar way we can **define n-th order conflicts** for each positive integer *n*. This is similar to the notion of n-th order impacts defined in [Bohner&Arnold96b], defined in the context of impact analysis.

How severe a conflict is can be seen by looking at its order. A first-order conflict is more severe than a second-order conflict, since there is already a problem by only looking at the direct dependencies. On the other hand, it is also important to detect higher-order conflicts, precisely because they are much harder to find by hand. Even automatically it will not be very efficient because the transitive closure needs to be calculated. Moreover, using the transitive closure could lead to a combinatorial explosion of evolution conflicts.

IV 4.5.4 Evolution Conflict Table

We can now set up a complete evolution conflict table which covers all possible situations. Because the evolution conflicts do not make a distinction between *Refinement* and *EdgeRetyping* (or even *Coarsening* when we consider the discussion in the previous subsection), we will use the following shortcut notation:

> ***ChangeEdge(e,u,v,τ)*** denotes either *Refinement(e,u,v,τ)*, *EdgeRetyping(e,u,v,τ₁,τ)* (for any τ_1) or *Coarsening(e,u,v,τ)*.

	Extension *(u,υ)*	*Cancellation* *(u,υ)*	*ChangeEdge* *(f,u,w,τ)*	*ChangeEdge* *(f,w,u,τ)*	*ChangeEdge* *(f,u,v,τ)*	*ChangeEdge* *(f,v,u,τ)*	*NodeRetype* *(u,ω,υ)*
ChangeEdge *(e,u,v,φ)*	×	×	EC_3 $(v\neq w)$	EC_1 $(v\neq w)$	EC_2 $(e\neq f)$	EC_5	EC_7
ChangeEdge *(e,v,u,φ)*	×	×	EC_1 $(v\neq w)$	EC_4 $(v\neq w)$	EC_5	EC_2 $(e\neq f)$	EC_6

Table 2: Evolution conflicts for primitive contract types

In practical situations, it might be desirable to make a distinction between conflicts that involve *Refinement*, *EdgeRetyping* or *Coarsening*, so that evolution conflicts can be ignored in some of these cases, while not in others. This will be necessary in particular domain-specific customisations of the formalism. In order to make a distinction between a *Refinement* and an *EdgeRetyping* by merely looking at the result graph, the modification tag should be enhanced, so that it does not only contain the name of the evolver/reuser, but also the kind of modification that has taken place.

IV 4.6 IMPLEMENTATION ISSUES

In this section we discuss some experiments, implementation issues and efficiency issues related to the detection of evolution conflicts as presented before. Readers that are interested in the formal aspects only can skip this section.

IV 4.6.1 Experiments

In order to test the formalism of primitive contract types and primitive reuse contracts, a PROLOG implementation has been made. The reason why we chose PROLOG is mainly because of its declarative nature. It allows us to detect evolution conflicts in almost exactly the same way as it was defined here.

Graphs are represented in the implementation in a very straightforward way. All nodes and edges of a graph are expressed as facts in the PROLOG database. New facts (nodes or edges) can be dynamically *asserted* (e.g., when performing an *Extension* or a *Refinement*) or *retracted* (e.g., when performing a *Cancellation* or a *Coarsening*). The primitive contract types (graph productions) are defined as PROLOG rules. The application preconditions and postconditions for each of the primitive contract types are equally expressed as PROLOG rules. Finally, checking of a graph pattern in a particular graph can be expressed in a very concise way by making use of the powerful *unification mechanism* of PROLOG.

The PROLOG implementation has been developed in parallel with the reuse contract formalism that is described in this dissertation. This was a very useful approach, since the practical implementation gave us new insight in the developed formalism and vice versa. Because it is only a prototype implementation to validate our ideas, we have not yet considered any efficiency issues.

The current implementation checks for applicability conflicts as well as evolution conflicts, and also allows us to express additional type constraints. Moreover, the implementation has been made in such a way that it can be easily customised to different application domains. For example, a filter is implemented that allows us to ignore particular evolution conflicts based on domain-specific information. More about this will be said in the next chapter.

The prototype implementation does not yet deal with transitive closure conflicts because of the efficiency problems involved when implementing them in a straightforward way. By using more sophisticated algorithms, and by relying on impact analysis techniques [Bohner&Arnold96b], the efficiency can be improved significantly.

What is also missing is a tool for visualising the underlying graphs, preferably with information hiding mechanisms that allow us to reduce the inherent complexity that arises when dealing with large software systems. A suitable candidate would be the encapsulation mechanism on hierarchical graphs that is proposed in [Engels&Schürr95].

IV 4.6.2 Modification Tags

In this subsection we discuss some technical issues regarding the modification tags that need to be taken into account when implementing a conflict detection tool.

In order to detect evolution conflicts by means of graph patterns, we have assumed until now that we started from an initial graph *G* in which the modification tags of all the nodes and edges were empty. If this is the case, conflicts can be detected if different modifiers add different modification tags to the nodes and edges they modified. As a result, after merging these independent modifications, we get a result graph *H* in which all modified or newly introduced nodes and edges are tagged, while all nodes or edges that remain untouched have an empty tag. The algorithm for detecting and resolving evolution conflicts in *H* would then go as follows:

Conflict Detection and Resolution Algorithm

(1) Detect all potential evolution conflicts in *H* using the graph pattern approach.

(2) Ask input from the software developer to determine which of these potential evolution conflicts are actual conflicts, and which are not.

(3) Store any evolution conflicts that should be ignored in a table for future reference.

(4) Resolve the actual evolution conflicts in a semi-automated way, thereby modifying graph *H*.[3]

(5) After this conflict resolution, use the graph pattern approach to detect if there are any remaining or new evolution conflicts in the modified graph. Ignore all conflicts in the table constructed in step (3). If there are no new conflicts, stop. If there are, go back to step (3).

The result of the algorithm presented above, which we will call H_0, can now be used as a new initial graph that can be evolved by different modifiers. However, it is no longer the case that all modification tags are empty. It is also not desirable to remove the modification tags in H_0, since then we would lose essential information about who made the earlier modifications. What we can do, however, is attach an additional flag (possibly containing a timestamp) to all modification tags in H_0, to indicate that all these modification tags have already been used before in the conflict detection algorithm. In this way, a

[3] In [Lippe&vanOosterom92], different strategies can be selected to solve conflicts: impose an order on the primitive modifications, delete some of the modifications, edit existing modifications or add new modifications. Since the last three strategies may introduce new conflicts, the conflict detection algorithm needs to be applied again in step (5).

distinction can be made between new and old modification tags. When merging different modifications of H_0 into a new result graph K, only the interactions between the new modification tags will be considered.

Each time a new modification is being made, the result graph becomes more complex, and will contain more nodes, edges and modification tags. This is even the case when nodes and edges are “removed”, since removing corresponds to giving a new type *«removed»* to the node or edge under consideration (see subsection IV 4.5.2). Obviously, this can make the graphs very large, so once in a while it is necessary to clean up the graph, by deleting all unnecessary information. More specifically, we could decide to actually delete all edges and nodes with type *«removed»* from the graph. If desired, we could also remove all modification tags from the graph during this step.

IV 4.6.3 Efficiency Issues

From a formal point of view, the approach of detecting graph patterns is usually easier to deal with than the approach where we need to find evolution conflicts by looking them up in the conflict table. The graph pattern approach is also more scalable because it does not rely on the primitive contract types that are involved. On the other hand, however, finding a graph pattern in an arbitrary graph is not always very efficient (although the search can be localised using the pullback of Definition 45). Hence, for efficiency reasons the approach with conflict tables might be preferred.

If we also want to detect transitive conflicts, we need to take the transitive closure graph into account. A first problem here is that calculating this transitive closure is a time-consuming process when using a straightforward implementation. However, more efficient algorithms have been developed over the years to counter this problem. A second and more important problem is that the number of edges in the transitive closure graph will be considerably higher than in the original graph. As a result, many more evolution conflicts will be detected. Sometimes there are so many evolution conflicts that they become unmanageable to deal with. Therefore, tools, techniques and heuristics should be developed that calculate only the most important evolution conflicts, or that allow us to reduce the evolution conflicts to a manageable number.

Although it is outside the scope of this dissertation to deal with this in more detail, it might be worthwhile to look at how existing impact analysis techniques deal with this problem [Bohner&Arnold96a]. Usually, these techniques make use of sophisticated search algorithms that take more factors into account than just plain dependencies. They can rely on predetermined semantics of specific node types and edge types, make use of heuristics that suggest which paths could be avoided, use stochastic probabilities to determine the likelihood of an impact, or even a combination of all these.

IV . 5 SUMMARY

In this chapter we provided a *formal foundation for reuse contracts*. By defining it on top of the formalism of labelled typed graphs and conditional graph rewriting we could rely on many known properties of these underlying formalisms. Besides providing a formal foundation for reuse contracts, this chapter also motivated the use of this formalism for supporting software evolution.

IV 5.1.1 Followed Approach

In order to formally detect undesired interactions when merging independent evolutions of the same software artifact, the following approach was followed:

- First, an orthogonal set of *primitive contract types* was defined in terms of *conditional productions*. These primitive contract types described the elementary modifications that can be made to a graph, and correspond to arbitrary evolutions of a software artifact.
- Next, *applicability conflicts* were defined to express syntactic incompatibilities when trying to merge independent evolutions of the same graph. To this end, the notion of *parallel independence* of graph productions was used. This characterisation was further refined by identifying all possible pairs of primitive contract types that can lead to an applicability conflict. This resulted in an *applicability conflict table*.
- By attaching applicability conditions to graphs or graph rewriting systems it became possible to restrict the possible modifications that could be made to an arbitrary software artifact. To achieve this, the concepts of *anticipated evolution conflict* and *evolution invariant* were introduced.
- In order to deal with semantic incompatibilities when merging parallelly independent evolutions of the same software artifact, *potential evolution conflicts* were defined theoretically in terms of a *pullback* construction. Again, this characterisation could be refined further by detecting the occurrence of particular *graph patterns* in the corresponding *pushout*. An alternative approach, where the conflicts were detected by identifying particular pairs of primitive contract types, resulted in an *evolution conflict table*.

Because the three approaches above – applicability conflicts, anticipated evolution conflicts and potential evolution conflicts – complement each other, they should all be combined in a single powerful and sophisticad conflict detection tool.

IV 5.1.2 Conservative Approach

An important feature of reuse contracts, or any other approach that tries to detect behavioural problems, is that it can only detect *potential* evolution conflicts. Indeed, according to [Rice53] any nontrivial property about the behaviour of a program is undecidable. Therefore, the only thing we can do is try to provide a safe approximation that generates conflict warnings in all situations where potentially a problem can arise.

A problem is that the number of generated conflict warnings can be very large in practice. Indeed, because we do not (yet) have any domain-specific knowledge, we detect an upper bound of everything that might go wrong. In Chapter VI we will customise the formal reuse contract formalism to different domains, and see that this allows us to remove many unnecessary conflict warnings because of the domain-specific semantics that is attached to particular types of nodes and edges. Also, domain-specific type constraints can be used to transform particular domain-independent evolution conflicts into domain-specific applicability conflicts.

In order to manage the large number of evolution conflicts that might still remain, we could sort the detected conflicts in order of importance. One possible way to do this is by making a distinction between first-order conflicts, second-order conflicts and higher-order conflicts if we take the transitive closure of edges into account. We can also resort to existing impact analysis techniques [Bohner&Arnold96a] to reduce the evolution conflicts to a manageable number.

IV 5.1.3 Relation to Previous Work

An important difference with previous work on reuse contracts [Steyaert&al96, Lucas97] is that our approach is completely *orthogonal and formal*.

- First, an explicit distinction is made between *applicability conflicts* and *evolution conflicts*. Applicability conflicts correspond to *structural* or *syntactic* inconsistencies, while evolution conflicts correspond to *behavioural* or *semantic* inconsistencies. An important difference between the two is that applicability conflicts are always conflicts: if a contract type is not applicable to a graph, it is impossible to generate a result graph. On the other hand, because of the conservative approach, evolution conflicts are only *potential conflicts*. It depends on the specific situation in which the conflict arises. Hence, one can only decide if an evolution conflict leads to an inconsistency in a semi-automated way. Feedback from the software developer is needed about how the conflict should be interpreted in order to resolve the problem.
- Second, an orthogonal set of primitive contract types is proposed. Because of the orthogonality, it is possible to give a *complete characterisation of all different kinds of conflicts* (applicability as well as evolution) that can occur. If suffices to compare all possible pairs of primitive contract types, and put them in a conflict table. This is important for tool support, as it allows us to detect evolution (and applicability) conflicts in a simple and efficient way. Of course, if we want to detect more sophisticated conflicts that involve the transitive closure, it will become less efficient.
- Another difference with [Lucas97] is that we are able to detect conflicts by looking for *graph patterns*. This approach is more scalable than using conflict tables, because we do not have to rely on primitive contract types.

IV 5.1.4 What's Next

Although this chapter formally addresses the basic ideas of the reuse contract approach, it still remains very primitive in practice. In the next chapter we will see that the fact of using a formal approach makes it easy to *scale up* to more complex situations. For example, we will be able to remove redundancy in an arbitrary sequence of evolution steps by means of a so-called *normalisation algorithm*. We will also discuss the impact on conflict detection when *composite contract types* are introduced. The influence of nesting will be investigated in more detail as well.

After having shown the scalability of our approach, we still need to validate its *domain-independence*. From an intuitive point of view, this should already be clear. The formalism only reasons about nodes, edges and types, which can be used as a basis for expressing any kind of software artifact. By customising the formal framework to different domains, we will actually validate this conviction.

V. Scalability of the Formalism

This chapter explains how the formal foundation of the previous chapter can be scaled up to deal with more complex situations, like arbitrary sequences of evolution steps and nesting. The impact of this scalability on the possible evolution conflicts is discussed.

V . 1 INTRODUCTION

In the previous chapter we have presented the basic ideas behind reuse contracts in a formal way. First, we gave a characterisation of the possible kinds of graph modifications by means of primitive contract types. Next, we discussed under which conditions two independent primitive reuse contracts could be serialised, and how applicability conflicts and evolution conflicts between incompatible primitive reuse contracts could be detected.

Despite these important results, the proposed formalism is still much too primitive to be useful in practice. Therefore, we will focus on some important *scalability issues* in this chapter. Indeed, as stated in chapter I, one of the aims of our thesis is to provide a formal foundation for reuse contracts which is *scalable*. Different scalability issues will be addressed in the different sections of this chapter.

- In section V . 2 we discuss how *primitive* contract types can be combined into *composite* contract types, which are basically sequences of more than one primitive contract type. As an important theoretical result, a normalisation algorithm is introduced to remove redundancy in arbitrary composite contract types, thus making the detection of conflicts easier and more efficient.
- Section V . 3 presents some useful predefined composite contract types that correspond to frequently occurring combinations of primitive contract types. In some cases, the intuitive meaning associated to these predefined composite contract types can be exploited to make the conflict detection more efficient and more flexible, e.g., by ignoring particular evolution conflicts in particular cases.
- Section V . 4 addresses an important problem of graphs, namely that the number of nodes and edges tends to become very large, so that the graphs become too complex to understand. Therefore, a nesting mechanism needs to be introduced. Because this extension is made at the level of graphs, it will influence the reuse contract framework that is defined on top of it. Fortunately, the required changes to the reuse contract framework turn out to be small. Some small modifications to the primitive contract types are needed, as well as new primitive and composite contract types. Also, some new evolution conflicts can be defined.
- Section V . 5 discusses an extension which needs to be made to the reuse contract formalism in order to be able to explicitly document reuse and evolution, and to deal with propagation of changes. The idea is that evolution and reuse can be represented explicitly as edges in a graph. To this end, the underlying graph formalism needs to be extended with the notion of *derived edges*.
- Finally, section V . 6 discusses some other extensions which could be made to the reuse contract approach, but which have not been investigated in detail because of time constraints.

V . 2 Composite Contract Types

V 2.1 Motivation

Scalability can be addressed in different ways. In this section we will focus on mechanisms that take a *sequence* of primitive reuse contracts, and try to reduce the complexity when detecting conflicts in this sequence. The basic idea is depicted in Figure 29, where an initial graph is modified through application of a sequence of primitive reuse contracts by a first reuser, while it is modified through a single primitive reuse contract by a second reuser. An even further extension of this situation would be to compare *two* sequences of contract types.

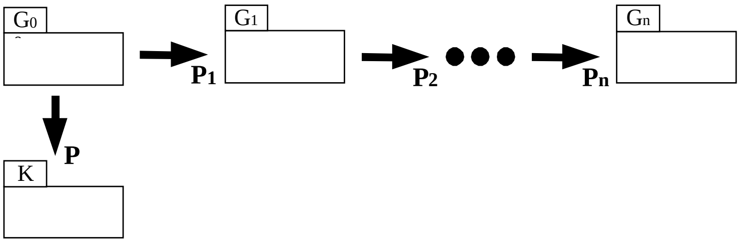


Figure 29: Composite Evolution Conflicts

One way to deal with sequences of primitive contract types is by introducing so-called *composite* contract types. While a sequence of *n* contract types requires *n-1* intermediate graphs to be calculated, a composite contract type combines all the modifications in one single production that behaves as an atomic whole. In this way, there is no need to generate any intermediate graphs, thus making the evolution process more efficient.

A second way to facilitate evolution is by distinguishing *intermediate conflicts* and *final conflicts*. When considering a sequence of primitive evolution steps, we sometimes temporarily want to ignore any inconsistencies. This can be because of two reasons. First, resolution of a conflict sometimes depends on information that is not yet available, so it would be unwise to force premature decisions. Second, some conflicts in the sequence may be removed later on, because the modification that gave rise to the conflict is again removed. In order to deal with this situation, we only need to consider final conflicts, i.e., conflicts that are still present after the evolution sequence has been applied in its entirety.

The problem stated above, where some intermediate conflicts become obsolete by modifications later in the sequence, is due to the fact that an arbitrary evolution sequence can contain a lot of *redundancy*. For example, upon evolution, a node might have been introduced and removed again, the same edge might have been retyped more than once, etc. In order to make the evolution process more comprehensible, we do not want to bother with all these intermediate details, but just want to know which basic modifications are required to modify the initial graph G_0 into the result graph G_n. As already mentioned in [Lippe&vanOosterom92], removing redundant transformations is attractive because it speeds up conflict detection (by compacting the evolution sequence) and it allows us to remove unnecessary conflicts. For this reason, we will define a *normalisation algorithm* that transforms an arbitrary sequence of primitive contract types into a *minimal* one where all redundant information is removed. An additional advantage is that it makes the evolution sequence more comprehensible, because all the primitive evolution steps are rearranged according to their contract type.

V 2.2 Composite Evolution Conflicts

V 2.2.1 Definition

Formally, a composite reuse contract is a sequence of primitive contract types applied one after the other.

Let G_0 and G_n be two graphs.
A **composite reuse contract** is a conditional derivation sequence of the form $G_0 \Rightarrow^+ G_n$, i.e.,
$G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ such that $n \geq 2$ and
$\forall i \in \{1..n\}$: P_i is a primitive contract type (and $G_{i-1} \Rightarrow_{Pi} G_i$ is a primitive reuse contract)

Definition 47: Composite reuse contract

From a practical point of view, the above definition does not give rise to a very efficient implementation of composite contract types. In order to calculate G_n from G_0 using a sequence of n primitive contract types, n-1 intermediate graphs need to be calculated. Therefore, we will introduce the notion of **composite contract type**, which calculates G_n from G_0 without needing any intermediate graphs. This has the additional advantage that the composite contract type behaves as an atomic action, similar to database transactions. Either all derivation steps are performed together (if the derivation sequence is applicable) or the graph is left in its original state (if the derivation sequence is not applicable).

Formally, this can be achieved by calculating the composite contract type from the sequence of primitive ones using the notion of composite production (Definition 35 of page 65). Property 9 then guarantees that each composite reuse contract has a corresponding composite contract type.

Let $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ be a composite reuse contract, consisting of a sequence of primitive contract types P_i. Then the composite production $P_1{*};P_2{*};\ldots;P_n{*}: G_0 \rightarrow G_n$ is called a **composite contract type**.

Definition 48: Composite contract type

V 2.2.2 Conflict Detection Algorithm

In the situation of Figure 29, we can detect applicability and evolution conflicts as before by consecutively comparing P with each P_i, thereby calculating an intermediate graph K_i at each step. This gives rise to the following straightforward algorithm:

Conflict detection algorithm

(i) Compare reuse contracts $G_0 \Rightarrow_P K$ and $G_0 \Rightarrow_{P1} G_1$. If they are *not* parallelly independent, one of the *applicability conflicts* of section IV . 3 is detected, and the algorithm stops. If they *are* parallelly independent, they are serialised ($G_0 \Rightarrow_{P1} G_1 \Rightarrow_P K_1$), leading to an intermediary result graph K_1. If one of the potential *evolution conflicts* of section IV 4.4 is detected, the algorithm stops.

(ii) Repeat step (i) with reuse contracts $G_1 \Rightarrow_P K_1$ and $G_1 \Rightarrow_{P2} G_2$. This gives an applicability or evolution conflict, or results in a new intermediary graph K_2 (obtained by the serialisation $G_1 \Rightarrow_{P2} G_2 \Rightarrow_P K_2$). In the latter case, repeat the process with $G_2 \Rightarrow_{P3} G_3$ and $G_2 \Rightarrow_P K_2$, and continue until an applicability or evolution conflict arises, or until the last reuse contract $G_{n-1} \Rightarrow_{Pn} G_n$ in the sequence has been compared with P, leading to a final result graph K_n.

In a CASE tool, this conflict detection algorithm could be supplemented with a semi-automated *conflict resolution algorithm*. Each time an application conflict arises, the software developer will be instructed to alter one or both modifications, assisted by the tool. Some conflicts like name clashes can be resolved automatically, while others require more work from the developer. Evolution conflicts can be resolved in a similar way. [Lippe&vanOosterom92] and [Mezini97] present a number of different strategies for solving conflicts.

V 2.2.3 Need for Normalisation

A disadvantage of the above conflict detection algorithm is that often it detects too many conflicts to be practical. The reason for this lies with the fact that a composite contract type can contain many redundant primitive contract types. This leads to the detection of a number of applicability and evolution conflicts that could be avoided by taking a more sophisticated approach. If we remove all redundant modifications first, many unnecessary intermediate conflicts can be avoided, giving rise to less actual conflicts and making the conflict detection process more efficient.

Let us take a look at two examples to illustrate this more clearly. They both start from an initial graph G_0 which is modified by primitive contract type P = *Refinement(e,u,v,τ)* on the one hand, and by a sequence of two primitive contract types P_1 and P_2 on the other hand:

- If P_1 = *Refinement(e,u,v,τ)* and P_2 = *Coarsening(e,u,v,τ)*, the conflict detection algorithm would stop after checking the first contract type P_1, since the combination of P and P_1 leads to an applicability conflict *AC5: Duplicate edge conflict*. Nevertheless, if we would first apply the composite sequence $P_1;P_2$ as a whole, followed by the primitive contract type P, the applicability conflict would not occur. Indeed, the modifications introduced by P_2 are undone by P_1.
- If P_1 = *Refinement(e,v,w,τ)* and P_2 = *Coarsening(e,v,w,τ)* we get a similar situation, except that the conflict detection algorithm now detects an evolution conflict *EC1: Reachability conflict*. Again, this conflict would be avoided if we would apply the composite sequence $P_1;P_2$ as a whole first, followed by the primitive contract type P afterwards.

From this discussion we can conclude that the conflict detection algorithm gives rise to two different kinds of conflicts. *Intermediate conflicts* are conflicts that are detected but do not lead to a problem if the sequence is applied as a whole. *Final conflicts* are *real* conflicts, in the sense that they are always detected. In order to improve the conflict detection algorithm so that it no longer detects unnecessary intermediate conflicts, there are two alternatives. The first one goes as follows:

Alternative conflict detection algorithm

(i) Apply the composite contract type $P_1*;P_2*;\ldots;P_n*$ as a whole to the initial graph G_0, leading to the result graph G_n.
(ii) Try to apply the primitive contract type P to this result graph G_n. If it fails, we have an applicability conflict, and the algorithm stops. If it succeeds, we get a new result graph K_n.
(iii) Detect all potential evolution conflicts in K_n using the graph pattern approach of section IV 4.4.3.

Although this new algorithm is very simple, it has two disadvantages. First, if an applicability conflict arises, it is difficult to see which of the primitive contract types P_i was the cause of this conflict, since all the primitive contract types have been applied as a whole. Secondly, detecting potential evolution conflicts by looking for graph patterns in the result graph is not always efficient. Therefore, we will now present a second alternative, which is only a small extension of the original conflict detection algorithm of section V 2.2.2.

The second alternative tries to avoid unnecessary intermediate conflicts by first performing a pre-processing phase to remove all redundant information in the composite reuse contract. The result is a "normalised" composite reuse contract, which no longer yields intermediate conflicts, but nevertheless leads to the same result graph as the original reuse contract. Assuming the existence of an adequate normalisation algorithm, the revised conflict detection algorithm would look as follows:

Revised conflict detection algorithm

(i) Transform the composite reuse contract $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ into a "normalised" version $G_0 \Rightarrow_{Q1} H_1 \Rightarrow_{Q2} \ldots \Rightarrow_{Qm} G_n$ (with $m \leq n$) which does not contain any redundant primitive contract types.
(ii) Apply the conflict detection algorithm of section V 2.2.2 to this normalised composite reuse contract and $G_0 \Rightarrow_P K$.

In section V 2.4 we will discuss in detail how this normalisation algorithm can be defined. To this end, we first need to introduce the notion of monotonous composite contract types.

V 2.3 Monotonous Composite Contract Types

In order to make a particular evolution sequence more comprehensible, it is useful to cluster all primitive contract types of the same kind. A sequence of primitive contract types of the same kind will be called a **monotonous composite contract type**, and we will show here that (except for *NodeRetypings* and *EdgeRetypings*) the order of the elements in such a monotonous sequence is irrelevant. In practice, we will deal with monotonous contract types as if they were atomic productions, by using the notion of *composite production* of Definition 35.

As part of the normalisation algorithm we will later see how an arbitrary sequence of primitive contract types can be reduced to a sequence of monotonous composite contract types.

Let $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ be a composite reuse contract with corresponding composite contract type $C_{Comp} = P_1{*};P_2{*};\ldots;P_n{*}$. C_{Comp} is called **monotonous** if all its primitive contract types are of the same kind. We can distinguish the following six cases:

A **composite extension** C_{Ext} is a monotonous composite contract type such that $\forall i \in \{1..n\}: \exists v_i \in NodeLabel: \exists \omega_i \in NodeType: P_i = Extension(v_i,\omega_i)$. A **composite cancellation** C_{Canc} is a monotonous composite contract type such that $\forall i \in \{1..n\}: \exists v_i \in NodeLabel: \exists \omega_i \in NodeType: P_i = Cancellation(v_i,\omega_i)$. In a similar way, we define **composite refinement** C_{Ref}, **composite coarsening** C_{Coars}, **composite node retyping** $C_{NodeRet}$ and **composite edge retyping** $C_{EdgeRet}$.

Definition 49: Monotonous composite contract types

Because a composite reuse contract is by definition a well-formed sequence of more primitive ones, the definitions above are restricted in the sense that they do not allow duplicates. For example, a composite extension cannot contain two primitive extensions that both introduce a new node with the same name. Similarly, a composite refinement cannot contain two primitive refinements that both introduce a new edge with the same name between the same two nodes.

Only in the case of a composite edge retyping and node retyping, duplicates are possible, since the type of a node or edge can be changed more than once. The underlying reason for this distinction between retypings and the other primitive contract types is that retypings are not orthogonal (see section IV 2.2.3). They can be expressed in terms of the four other ones.

The above discussion is captured in the following property:

If $G_0 \Rightarrow_{CExt} G_n$, then $\forall i \neq j$: if $P_i = Extension(v,\upsilon)$ and $P_j = Extension(w,\omega)$ then $v \neq w$.
If $G_0 \Rightarrow_{CCanc} G_n$, then $\forall i \neq j$: if $P_i = Cancellation(v, \upsilon)$ and $P_j = Cancellation(w, \omega)$ then $v \neq w$.
If $G_0 \Rightarrow_{CRef} G_n$, then $\forall i \neq j$: if $P_i = Refinement(e,v_1,v_2,\tau)$ and $P_j = Refinement(f,w_1,w_2,\phi)$ then $(e,v_1,v_2) \neq (f,w_1,w_2)$.
If $G_0 \Rightarrow_{CCoars} G_n$, then $\forall i \neq j$: if $P_i = Coarsening(e,v_1,v_2,\tau)$ and $P_j = Coarsening(f,w_1,w_2,\phi)$ then $(e,v_1,v_2) \neq (f,w_1,w_2)$.

Property 17: Well-formedness of a monotonous composite reuse contract

Proof:

(a) Let $C_{Ext} = P_1{*};P_2{*};\ldots;P_n{*}$. Suppose that $\exists\ j>i$ such that $P_i = Extension(v,\upsilon)$ and $P_j = Extension(v,\omega)$. Then C_{Ext} cannot be a composite extension, since the precondition $\{v \notin L_j\} \subseteq PreCond(P_j)$ is not satisfied, because $\{v \in L_i\} \subseteq PostCond(P_i)$. Moreover, v has not been removed somewhere in between P_i and P_j since C_{Ext} only contains extensions.
(b) The reasoning is analogous if C_{Canc} is a composite cancellation.
(c) Let $C_{Ref} = P_1{*};P_2{*};\ldots;P_n{*}$. Suppose that $\exists\ j>i$ such that $P_i = Refinement(e,v_1,v_2,\tau)$ and $P_j = Refinement(e,v_1,v_2,\phi)$. Then C_{Ref} cannot be a composite refinement, since the precondition $\{(e,v_1,v_2) \notin L_j\} \subseteq PreCond(P_j)$ is not satisfied, because $\{(e,v_1,v_2) \in L_i\} \subseteq PostCond(P_i)$. Moreover, (e,v_1,v_2) has not been removed somewhere in between P_i and P_j since C_{Ref} only contains refinements.
(d) The reasoning is analogous if C_{Coars} is a composite coarsening.

An important result states that composite extensions (resp. cancellations, refinements and coarsenings) are sequentially independent, i.e., the order in which the productions in the sequence are applied is irrelevant:

If $C_{Comp} = P_1*;P_2*;\ldots;P_n*$ is a composite extension, cancellation, refinement or coarsening, then
(i) $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ is a sequentially independent conditional derivation sequence.
(ii) The order of applying the primitive contract types P_i in the sequence C_{Comp} is arbitrary, and always leads to the same result.
(iii) There are no applicability conflicts between any two primitive contract types P_i and P_j of C_{Comp}.
(iv) Each ordering leads to the same potential evolution conflicts.

Property 18: Independence of order in monotonous composite contract types

Proof:

We show the proof when C_{Comp} corresponds to a sequence of only two primitive contract types, i.e., $C_{Comp} = P_1*;P_2*$. If there are more than two productions, we can change their order pairwise.

(i) The proof needs to be split in four parts, for composite extension, composite refinement, composite cancellation and composite coarsening respectively.

- Let $C_{Comp}=C_{Ext}$, i.e., $P_1 = Extension(v_1,\upsilon)$ and $P_2 = Extension(v_2,\omega)$ with $v_1 \neq v_2$. Using Definition 39, $G_1 \Rightarrow_{P2} G_2$ is weakly sequentially independent of $G_0 \Rightarrow_{P1} G_1$, because $PreCond(P_2)=\{v_2 \notin L_2\}$ is preserved by P_1 in G_0, since P_1 does not remove any nodes. Using the same definition, $G_0 \Rightarrow_{P1} G_1$ is weakly parallelly independent of $G_0 \Rightarrow_{P2} H$, because $PreCond(P_1)=\{v_1 \notin L_1\}$ is preserved by P_2 in H, since P_2 does not introduce the node v_1 the absence of which is required by P_1. It only introduces a node $v_2 \neq v_1$. Using Definition 31, we can conclude that $G_0 \Rightarrow_{P1} G_1$ and $G_1 \Rightarrow_{P2} G_2$ are sequentially independent. As a result, $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} G_2$ is a sequentially independent conditional derivation sequence.
- Let $C_{Comp}=C_{Ref}$, i.e., $P_1 = Refinement(e,v_1,v_2,\tau)$ and $P_2 = Refinement(f,w_1,w_2,\phi)$ with $(e,v_1,v_2) \neq (f,w_1,w_2)$. $G_1 \Rightarrow_{P2} G_2$ is weakly sequentially independent of $G_0 \Rightarrow_{P1} G_1$, because $PreCond(P_2)=\{w_1 \in L_2,\ w_2 \in L_2,\ (f,w_1,w_2) \notin L_2\}$ is preserved by P_1 in G_0, since P_1 does not remove w_1 or w_2 and does not introduce an edge (f,w_1,w_2). Similarly, $G_0 \Rightarrow_{P1} G_1$ is weakly parallelly independent of $G_0 \Rightarrow_{P2} H$, because $PreCond(P_1)=\{v_1 \in L_1,\ v_2 \in L_1,\ (e,v_1,v_2) \notin L_1\}$ is preserved by P_2 in H, since P_2 does not remove v_1 or v_2 and does not introduce an edge (e,v_1,v_2). Using both results, we can conclude that $G_0 \Rightarrow_{P1} G_1$ and $G_1 \Rightarrow_{P2} G_2$ are sequentially independent. As a result, $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} G_2$ is a sequentially independent conditional derivation sequence.
- If $C_{Comp}=C_{Canc}$, the proof is similar to the first part. If $C_{Comp}=C_{Coars}$, the proof is similar to the second part.

(ii) Because $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} G_2$ is a sequentially independent derivation sequence (see (i)), we can apply the confluence property, which states that the order of the productions is irrelevant, and always leads to the same result graph G_2.

(iii) Because $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} G_2$ is a sequentially independent derivation sequence, and P_1 and P_2 are injective (Property 12), the confluence property guarantees that $\exists$ unique graph H such that $G_0 \Rightarrow_{P1} G_1$ and $G_0 \Rightarrow_{P2} H$ are parallelly independent. Consequently, according to Definition 43, they do not lead to any applicability conflicts.

(iv) If $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} G_2$ has potential evolution conflicts, then $G_0 \Rightarrow_{P2} H \Rightarrow_{P1} G_2$ has exactly the same evolution conflicts, since evolution conflicts can be detected by looking at the result graph G_2 only, and the result graph G_2 is the same in both cases because of the uniqueness in the confluence property.

Again, Property 18 is not valid for composite edge retypings and composite node retypings, for the same reason that Property 17 is not valid. This can easily be seen in the following counterexample. Take P_1=$NodeRetyping(u,\omega_1,\omega_2)$, P_2=$NodeRetyping(u,\omega_2,\omega_1)$ and P_3=$NodeRetyping(u,\omega_1,\omega_3)$. While the given order is valid if we start from a graph G with $(u,\omega_1) \in G$, it is clear that any other order immediately leads to an applicability conflict. A similar reasoning can be made for *EdgeRetypings*.

An important corollary of Property 18 states that in the case of composite extensions, cancellations, coarsenings and refinements, sequential composition coincides with parallel composition:

Let $G_0 \Rightarrow_{CComp} G_n$ be a composite reuse contract with $C_{Comp} = P_1*;P_2*;…;P_n*$ a composite extension, cancellation, coarsening or refinement. Then $G_0 \Rightarrow_{CComp} G_n$ **coincides** with the atomic **parallel composition** $G_0 \Rightarrow_{P1+P2+…+Pn} G_n$.

Corollary 1: Sequential versus parallel composition

Indeed, in the considered cases all the primitive contract types P_i are injective, and the derivation sequence $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} … \Rightarrow_{Pn} G_n$ is sequentially independent. Hence, Property 5 can be used to find the unique parallel derivation $G_0 \Rightarrow_{P1+P2+…+Pn} G_n$.

V 2.4 Normalisation of Composite Contract Types

In this section we define the normalisation algorithm needed for removing redundancy in an arbitrary sequence of primitive contract types. An example of such a redundancy is *Extension(v,υ)* followed by *Cancellation(v,υ)*. If we want to remove these redundant contract types, the main problem is that they do not necessarily follow each other immediately. Often, a number of other primitive contract types will occur in between. As a result, before the redundant contract types can be removed, they first have to be brought closer together by swapping them with their preceding or succeeding contract types. Such swapping is only allowed if the contract types are *sequentially independent*. For this reason, we first need to investigate under which conditions primitive contract types are sequentially independent. This will be done in subsection V 2.4.1.

As a next step, we need to determine all possible situations in which two subsequent contract types are redundant. A distinction is made between *redundant contract types*, which are discussed in subsection V 2.4.2, and *absorbing contract types*, which are discussed in subsection V 2.4.4.

Finally, subsection V 2.4.6 gives the exact details of the normalisation algorithm.

V 2.4.1 Sequential Dependence of Primitive Contract Types

We are interested in determining which of the primitive contract types are mutually sequentially independent. In Property 18 we already partially solved this question for composite extensions, refinements, cancellations and coarsenings (which are always sequentially independent). We will now investigate in which cases combinations of *different* primitive contract types are sequentially independent.

For this, it suffices to investigate **weakly sequential independence** of two primitive contract types P_1 and P_2 in a composite reuse contract $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} G_2$, according to Definition 31 and Definition 39. Parallel independence has already been investigated in section IV . 3 , when discussing applicability conflicts. The results were summarised in Table 1 of page 88.

(a) P_2 = *Extension(v,υ)* is weakly sequentially independent of P_1 if $PreCond(P_2)=\{v \notin L_2\}$ is preserved by P_1 in G_0. This is **not** the case if P_1 = *Cancellation(v,ω)*.

(b) P_2 = *Cancellation(v,υ)* is weakly sequentially independent of P_1 if $PreCond(P_2)=\{(v,\upsilon) \in L_2, Adjacent_{L2}(v)=\emptyset\}$ is preserved by P_1 in G_0. This is **not** the case if P_1 = *Extension(v,ω)*, because the precondition $(v,\upsilon) \in L_2$ is not preserved in G_0. It is also **not** the case if P_1 = *NodeRetyping(v,ω,υ)* because the precondition $type_{L2}(v)=\upsilon$ is not preserved in G_0. A final problem occurs if P_1 = *Coarsening(e,v,w,τ)* or P_1 = *Coarsening(e,w,v,τ)* since the precondition $Adjacent_{L2}(v)=\emptyset$ is not preserved in G_0.

(c) P_2 = *Refinement(e,v,w,τ)* is weakly sequentially independent of P_1 if $PreCond(P_2)=\{v \in L_2, w \in L_2, (e,v,w) \notin L_2\}$ is preserved by P_1 in G_0. This is **not** the case if P_1 = *Extension(v,ω)* or P_1 = *Extension(w,ω)*, breaking the precondition $v \in L_2$ or $w \in L_2$. Similarly, if P_1 = *Coarsening(e,v,w,ϕ)* the precondition $(e,v,w) \notin L_2$ is not preserved in G_0.

(d) P_2 = *Coarsening(e,v,w,τ)* is weakly sequentially independent of P_1 if $PreCond(P_2)=\{v \in L_2, w \in L_2, (e,v,w,\tau) \in L_2\}$ is preserved by P_1 in G_0. If P_1 = *Refinement(e,v,w,τ)*, the precondition $(e,v,w) \in L_2$ is not preserved in G_0. If P_1 = *EdgeRetyping(e,v,w,ϕ,τ)*, the precondition $type_{L2}(e,v,w)=\tau$ is not preserved in G_0. Note that the preconditions $v \in L_2$ and $w \in L_2$ always hold in G_0, because P_1 cannot be an *Extension* with *v* or *w*, since G_1 contains an edge *e* between *v* and *w*.

(e) $P_2 = NodeRetyping(v,\upsilon,\upsilon_2)$ is weakly sequentially independent of P_1 if $PreCond(P_2)=\{(v,\upsilon)\in L_2\}$ is preserved by P_1 in G_0. This is **not** the case if $P_1 = Extension(v,\upsilon)$ or $P_1 = NodeRetyping(v,\omega,\upsilon)$, because the precondition $type_{L2}(v)=\upsilon$ is not preserved in G_0.

(f) $P_2 = EdgeRetyping(e,v,w,\tau,\tau_2)$ is weakly sequentially independent of P_1 if $PreCond(P_2)=\{v\in L_2, w\in L_2, (e,v,w,\tau)\in L_2\}$ is preserved by P_1 in G_0. If $P_1 = Refinement(e,v_1,v_2,\tau)$, the precondition $(e,v,w)\in L_2$ is not preserved in G_0. If $P_1 = EdgeRetyping(e,v_1,v_2,\phi,\tau)$, the precondition $type_{L2}(e,v,w)=\tau$ is not preserved in G_0. Note that the preconditions $v\in L_2$ and $w\in L_2$ always hold in G_0, because P_1 cannot be an *Extension* with v or w, since G_1 contains an edge e between v and w.

All these results are summarised in Table 3, which can be considered as an addition to Table 1. Fields in grey correspond to applicability conflicts (as discussed earlier), while fields in white correspond to *sequential dependencies*, i.e., situations where the order of application cannot be changed because P_2 sequentially depends on P_1. For example, $P_1 = Refinement(f,w_1,w_2,\phi)$ and $P_2 = Extension(v,\upsilon)$ corresponds to a grey field mentioning $(w_1=v \text{ or } w_2=v)$. This means that under these conditions P_2 is not applicable after P_1. Vice versa, $P_1 = Extension(w,\omega)$ and $P_2 = Refinement(e,v_1,v_2,\tau)$ corresponds to a white field mentioning $(v_1=w \text{ or } v_2=w)$. This indicates that under these conditions P_2 sequentially depends on P_1. Indeed, without the *Extension* with w, the *Refinement* cannot take place. In both cases, if the condition is **not** satisfied, then P_1 and P_2 are **sequentially independent**, and the order in which they appear is irrelevant. In the fields with √, there are no problems at all, so those cases are both sequentially and parallelly independent.

<table>
<tr><th>P_2
P_1</th><th>Extension
(v,υ)</th><th>NodeRetyping
(v,υ,υ_2)</th><th>Cancellation
(v,υ)</th><th>Refinement
(e,v_1,v_2,τ)</th><th>EdgeRetyping
(e,v_1,v_2,τ,τ_2)</th><th>Coarsening
(e,v_1,v_2,τ)</th></tr>
<tr><td rowspan="2">Extension
(w,ω)</td><td rowspan="4">$v=w$</td><td colspan="2">$v=w$ and $\upsilon=\omega$</td><td rowspan="2">$v_1=w$ or $v_2=w$</td><td colspan="2" rowspan="2">$v_1=w$ or $v_2=w$</td></tr>
<tr><td colspan="2">$v=w$ and $\upsilon\neq\omega$</td></tr>
<tr><td rowspan="2">NodeRetyping
(w,ω,ω_2)</td><td colspan="2">$v=w$ and $\upsilon=\omega_2$</td><td colspan="3" rowspan="2">√</td></tr>
<tr><td colspan="2">$v=w$ and $\upsilon\neq\omega_2$</td></tr>
<tr><td>Cancellation
(w,ω)</td><td>$v=w$</td><td colspan="2">$v=w$</td><td colspan="3">$v_1=w$ or $v_2=w$</td></tr>
<tr><td rowspan="2">Refinement
(f,w_1,w_2,ϕ)</td><td rowspan="5">$w_1=v$ or $w_2=v$</td><td rowspan="5">√</td><td rowspan="4">$w_1=v$ or $w_2=v$</td><td rowspan="4">$(e,v_1,v_2)=$
(f,w_1,w_2)</td><td colspan="2">$(e,v_1,v_2)=(f,w_1,w_2)$ and $\tau=\phi$</td></tr>
<tr><td colspan="2">$(e,v_1,v_2)=(f,w_1,w_2)$ and $\tau\neq\phi$</td></tr>
<tr><td rowspan="2">EdgeRetyping
(f,w_1,w_2,ϕ,ϕ_2)</td><td colspan="2">$(e,v_1,v_2)=(f,w_1,w_2)$ and $\tau=\phi_2$</td></tr>
<tr><td colspan="2">$(e,v_1,v_2)=(f,w_1,w_2)$ and $\tau\neq\phi_2$</td></tr>
<tr><td>Coarsening
(f,w_1,w_2,ϕ)</td><td>$w_1=v$ or $w_2=v$</td><td>$(e,v_1,v_2)=$
(f,w_1,w_2)</td><td colspan="2">$(e,v_1,v_2)=(f,w_1,w_2)$</td></tr>
</table>

Table 3: Sequential dependence of primitive contract types

In the remainder of this section, the results in this table will be needed to prove the normalisation algorithm.

V 2.4.2 Redundant Primitive Contract Types

We will now elaborate those situations where two subsequent (sequentially dependent) primitive contract types are each others inverse (as defined in Definition 42). If this is the case, we call them *redundant*.

> Let $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ be a composite reuse contract. A pair of subsequent primitive contract types (P_i, P_{i+1}) is **redundant** if $P_i = Inverse(P_{i+1})$ (or vice versa).

Definition 50: Redundant primitive contract types

According to Property 14 of page 83, the six cases of redundant primitive contract types are:

- $(Extension(v,\omega)), Cancellation(v,\omega))$ and vice versa
- $(Refinement(e,v,w,\tau), Coarsening(e,v,w,\tau))$ and vice versa
- $(NodeRetyping(v,\upsilon,\omega), NodeRetyping(v,\omega,\upsilon))$

- *(EdgeRetyping(e,v,w,τ,ϕ), EdgeRetyping(e,v,w,ϕ,τ))*

These pairs of contract types are called redundant because they can be removed in a composite reuse contract without influencing the result, as shown in the following property:

Let $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} G_2$ be a composite reuse contract. If (P_1, P_2) is redundant, then $G_0{=}G_2$.

Property 19: Removing redundant primitive contract types

Proof:

Because (P_1, P_2) is redundant, $P_2{=}Inverse(P_1)$. According to Definition 42, $PostCond(P_1) = PreCond(P_2)$ so all application preconditions of P_2 are valid. Moreover, $G_1 \Rightarrow_{Inverse(P1)} G_0$. Hence, $G_1 \Rightarrow_{P2} G_0$. Consequently, $G_2{=}G_0$, because $G_1 \Rightarrow_{P2} G_2$, and the result of applying a primitive contract type is unique because we work with injective label-preserving morphisms.

As a corollary, Property 19 can be extended to remove redundant primitive contract types in composite reuse contracts of arbitrary length.

Removing *redundant* pairs of primitive contract types (P_i, P_{i+1}) in a composite reuse contract $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ leads to a *unique* reduced composite reuse contract
$G_0 \Rightarrow_{P1} \ldots \Rightarrow_{Pi-1} G' \Rightarrow_{Pi+2} \ldots \Rightarrow_{Pn} G_n$ (with $G' = G_{i-1} = G_{i+1}$).

Corollary 2: Removing redundant primitive contract types

Proof:

(a) From Property 19 we know that $G_{i-1} = G_{i+1}$, since (P_i, P_{i+1}) are redundant. As a result, we can replace the original composite reuse contract by a new one $G_0 \Rightarrow_{P1} \ldots \Rightarrow_{Pi-1} G' \Rightarrow_{Pi+2} \ldots \Rightarrow_{Pn} G_n$ with the same result graph. It suffices to take $G' = G_{i-1} = G_{i+1}$.

(b) To prove uniqueness of the reduced composite reuse contract, we have to consider the situation $G_{i-1} \Rightarrow_{Pi} G_i \Rightarrow_{Pi+1} G_{i+1} \Rightarrow_{Pi+2} G_{i+2}$ where both (P_i, P_{i+1}) and (P_{i+1}, P_{i+2}) are redundant. In that case, the result of removing a pair of redundant contract types is either $G_{i-1} \Rightarrow_{Pi+2} G_{i+2}$ or $G_{i-1} \Rightarrow_{Pi} G_{i+2}$ according to **(a)**. However, $P_i = Inverse(P_{i+1})$ and $P_{i+1} = Inverse(P_{i+2})$, so $P_i = Inverse(Inverse(P_{i+2})) = P_{i+2}$ because of Property 13.

Because, after removing all redundant contract types in a composite reuse contract $G \Rightarrow_{CComp} H$, the result graph H remains the same, one could *incorrectly* conclude that the final evolution conflicts in the reduced composite reuse contract are *the same* as in the original one. In some cases there will be *fewer* evolution conflicts in the reduced composite reuse contract than in the original one, as can be seen in the following example.

Suppose that $G \Rightarrow_{CComp} H$ contains a primitive contract type $P_i{=}Cancellation(v,\omega)$ followed afterwards by primitive contract type $P_j{=}Extension(v,\omega)$. In other words, a node with label v and type ω is deleted somewhere in the sequence, and is reintroduced later on. Since both contract types P_i and P_j are each others inverse, they are redundant and will be removed. As a result, the reduced sequence no longer shows that a change has been made to v (since the redundant contract types have been removed). Hence evolution conflicts regarding changes to v (such as *EC6: Inconsistent target conflict* and *EC7: Inconsistent source conflict*) will *not* be detected. When looking at the composite reuse contract $G \Rightarrow_{CComp} H$ without removing redundant pairs, these final evolution conflicts *will* be detected using the graph pattern approach of section IV 4.4. More specifically, although in $G \Rightarrow_{CComp} H$ both G and H would contain a node v with the same type ω, the fact that a modification has been made to v somewhere during the evolution process can be seen because v will contain a modification tag $\{\rho_1\}$ in H (if ρ_1 is the reuser that made the modification), while the modification tag of v in G is empty. Consequently, an evolution conflict will still be detected although there seem to be no apparent changes to v.

Of course, one could wonder if it is really necessary to detect this kind of conflict in the case of redundant contract types. We claim that the answer is yes, since we do not know for sure if the reintroduction of v by P_j really corresponds to the *same* node, or an entirely new node that *accidentally* has the same label v. In the latter case, we have a very likely source of conflicts, since the independent contract type P of Figure 27 can make a modification involving v, thereby assuming that the old version of v is still used, while actually a modification of the new version of v is being made!

Observe that the reasoning above does not only hold for the specific example of *Cancellation* followed by *Extension*, but is valid in general for any pair of redundant contract types.

Because the above discussion could lead one to believe that the approach of removing redundant contract types is useless, we will show how we can deal with the above problems by introducing so-called "preserving" contract types.

V 2.4.3 Preserving Contract Types

How can we ensure that, after removing all redundant contract types, we obtain the same final evolution conflicts as before? The solution is to introduce new primitive contract types that do not make any changes to a graph, but simply specify that a node (or edge) has been modified in a preserving way. For example, a $Cancellation(v,\omega)$ followed by an $Extension(v,\omega)$ will be replaced by a $PreserveNode(v,\omega)$, indicating a change has been made to v without having an effect in the result graph H. Because of this new primitive contract type, final evolution conflicts will be detected, since the node v will contain a modification tag $\{\rho_1\}$ in H.

The opposite situation, where an $Extension(v,\omega)$ is followed by a $Cancellation(v,\omega)$, can be replaced by a *PreserveNode(v,«removed»)*, indicating that node v was absent before the two modifications, and remains absent after the two modifications. Formally, this can be expressed by means of a negative application condition that must be preserved.

In order to deal with all possible situations, we only need to introduce two **preserving contract types**, i.e., primitive contract types that do not modify anything to a graph.

A **node preservation** P = ***PreserveNode(v,ω)*** is a primitive contract type $P: L \rightarrow L$ with $PreCond(P) = \{(v,\omega)\in L\} = PostCond(P)$. If ω = *«removed»*, then $PreCond(P) = \{v \notin L\} = PostCond(P)$.
An **edge preservation** P = ***PreserveEdge(e,v,w,τ)*** is a primitive contract type $P: L \rightarrow L$ with $PreCond(P) = \{(e,v,w,\tau)\in L\} = PostCond(P)$. If τ = *«removed»*, then $PreCond(P) = \{(e,v,w)\notin L\} = PostCond(P)$.
A **composite node preservation** $C_{NodePres} = P_1{}^*;P_2{}^*;\ldots;P_n{}^*$ is a monotonous composite contract type such that $\forall i\in\{1..n\}\colon \exists v_i\in NodeLabel\colon \exists \omega_i\in NodeType\colon P_i=PreserveNode(v_i,\omega_i)$. A **composite edge preservation** $C_{EdgePres} = P_1{}^*;P_2{}^*;\ldots;P_n{}^*$ is a monotonous composite contract type such that $\forall i\in\{1..n\}\colon \exists e_i\in EdgeLabel\colon \exists v_i,\, w_i\in NodeLabel\colon \exists \tau_i\in EdgeType\colon P_i=PreserveEdge(e_i,v_i,w_i,\tau_i)$.

Definition 51: Preserving contract types

The process of removing pairs of redundant primitive contract types, as explained in section V 2.4.2, is then modified to the process of replacing a redundant pair by a preserving contract type:

- Replace $(Cancellation(v,\omega)), Extension(v,\omega))$ by $PreserveNode(v,\omega)$
- Replace $(Extension(v,\omega), Cancellation(v,\omega))$ by *PreserveNode(v,«removed»)*
- Replace $(Coarsening(e,v,w,\tau), Refinement(e,v,w,\tau))$ by $PreserveEdge(e,v,w,\tau)$
- Replace $(Refinement(e,v,w,\tau), Coarsening(e,v,w,\tau))$ by *PreserveEdge(e,v,w,«removed»)*
- Replace $(NodeRetyping(v,\upsilon,\omega), NodeRetyping(v,\omega,\upsilon)$ by $PreserveNode(v,\upsilon)$
- Replace $(EdgeRetyping(e,v,w,\tau,\phi), EdgeRetyping(e,v,w,\phi,\tau))$ by $PreserveEdge(e,v,w,\tau)$

All the cases enumerated above correspond to pairs of sequentially *dependent* contract types. Therefore, this definition can be considered as a special case of Definition 48 of page 110, where a sequence of contract types was combined into a single *composite contract type*. More specifically, the absorption contract type P_i' **coincides** with the composite contract type $P_i;P_{i+1}$ of Definition 35. Because we work with *conditional productions*, we need to take the application conditions into account as well. More specifically, $PostCond(P_i;P_{i+1}) = PostCond(P_{i+1})$, and $PreCond(P_i;P_{i+1})$ is obtained from $PreCond(P_i)$ and by anticipating all postconditions of P_i and all preconditions of P_{i+1}. This leads us to the following property which is an extension of Corollary 2 to include preservation of final evolution conflicts:

Replacing *redundant* pairs of primitive contract types (P_i, P_{i+1}) by a preserving contract type P_i' in a composite reuse contract $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \dots \Rightarrow_{Pn} G_n$ leads to a *unique* reduced composite reuse contract $G_0 \Rightarrow_{P1} \dots \Rightarrow_{Pi-1} G_{i-1} \Rightarrow_{Pi'} G_{i+1} \Rightarrow_{Pi+2} \dots \Rightarrow_{Pn} G_{n.}$
Moreover, this reduced reuse contract has the same final evolution conflicts as the original one.

Property 20: Replacing redundant contract types

Proof:

To prove this property, we only need to show that, for all pairs (P_i, P_{i+1}) of redundant primitive contract types, $P_i' = P_i;P_{i+1}$. Since replacing a sequence of contract types by a composite contract type does not essentially change anything to the composite reuse contract, and because of the confluency property, all other results immediately follow.

- If $(P_i, P_{i+1}) = (Cancellation(v,\omega), Refinement(v,\omega))$ then $PreCond(P_i)=\{(v,\omega)\in L_i\}$ and $PostCond(P_i)=\{v\notin R_i\}$. Similarly, $PreCond(P_{i+1})=\{v\notin L_{i+1}\}$ and $PostCond(P_{i+1})=\{(v,\omega)\in R_{i+1}\}$. Applying both contract types sequentially leads to a new contract type $P_i' = P_i;P_{i+1}$ with $PreCond(P_i')=\{(v,\omega)\in L_i\}=PostCond(P_i')$. Consequently, $P_i'=PreserveNode(v,\omega)$.
- A similar reasoning can be made for the three other cases.

Because the result graph G_n remains the same, and because each contract type P_i that modifies a node or edge in the original sequence has a corresponding contract type P_i' that does the same in the reduced sequence, it is clear that the reduced reuse contract has the same final evolution conflicts as the original one.

V 2.4.4 Absorbing Primitive Contract Types

We will now try to generalise the previous property a bit more. If $P_i = Cancellation(v,\omega)$ then all application conditions for $P_{i+1} = Extension(v,\upsilon)$ are satisfied, even though P_{i+1} is not precisely the inverse of P_i. Actually, the result of applying P_{i+1} after P_i can be described more simply in terms a new primitive contract type $P_i' = NodeRetyping(v,\omega,\upsilon)$. We say that P_i' has **absorbed** P_i and P_{i+1}. A similar reasoning can be made when $P_i = Coarsening(e,v,w,\phi)$ and $P_{i+1} = Refinement(e,v,w,\tau)$. In that case, both primitive contract types can be **absorbed** in an edge retyping $P_i' = EdgeRetyping(e,v,w,\phi,\tau)$. Another similar situation appears when $P_i = Extension(v,\upsilon)$ and $P_{i+1} = NodeRetyping(v,\upsilon,\omega)$. In this case, the node retyping can be absorbed in the extension, leading to a new extension $P_i' = Extension(v,\omega)$.

Obviously, the cases of redundant pairs can also be considered as being absorbed in a preserving contract type. All possible absorptions are summarised and formalised in the following definition and property.

Let $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \dots \Rightarrow_{Pn} G_n$ be a composite reuse contract. A pair of subsequent primitive contract types (P_i, P_{i+1}) is **absorbing**, with a corresponding **absorption contract type** P_i' in the following cases:

- (P_i, P_{i+1}) are redundant and $P_i' = P_i;P_{i+1}$ is a preserving contract type

- $(P_i, P_{i+1}) = (Cancellation(v,\omega), Extension(v,\upsilon))$ and $P_i' = NodeRetyping(v,\omega,\upsilon)$
- $(P_i, P_{i+1}) = (Extension(v,\upsilon), NodeRetyping(v,\upsilon,\omega))$ and $P_i' = Extension(v,\omega)$
- $(P_i, P_{i+1}) = (NodeRetyping(v,\upsilon,\omega), Cancellation(v,\omega))$ and $P_i' = Cancellation(v,\upsilon)$
- $(P_i, P_{i+1}) = (NodeRetyping(v,\upsilon_1,\upsilon_2), NodeRetyping(v,\upsilon_2,\upsilon_3))$ with $\upsilon_1\neq\upsilon_3$ and $P_i' = NodeRetyping(v,\upsilon_1,\upsilon_3)$
- $(P_i, P_{i+1}) = (Coarsening(e,v,w,\phi), Refinement(e,v,w,\tau))$ and $P_i' = EdgeRetyping(e,v,w,\phi,\tau)$
- $(P_i, P_{i+1}) = (Refinement(e,v,w,\tau), EdgeRetyping(e,v,w,\tau,\phi))$ and $P_i' = Refinement(e,v,w,\phi)$
- $(P_i, P_{i+1}) = (EdgeRetyping(e,v,w,\tau,\phi), Coarsening(e,v,w,\phi))$ and $P_i' = Coarsening(e,v,w,\tau)$
- $(P_i, P_{i+1}) = (EdgeRetyping(e,v_1,v_2,\tau_1,\tau_2), EdgeRetyping(e,v_1,v_2,\tau_2,\tau_3))$ with $\tau_1\neq\tau_3$ and $P_i' = EdgeRetyping(e,v_1,v_2,\tau_1,\tau_3)$

- $(P_i, P_{i+1}) = (Extension(v,\omega), PreserveNode(v,\omega))$ and $P_i' = P_i$
- $(P_i, P_{i+1}) = (Cancellation(v,\omega), PreserveNode(v,«removed»))$ and $P_i' = P_i$
- $(P_i, P_{i+1}) = (Refinement(e,v,w,\tau), PreserveEdge(e,v,w,\tau))$ and $P_i' = P_i$
- $(P_i, P_{i+1}) = (Coarsening(e,v,w,\tau), PreserveEdge(e,v,w,«removed»))$ and $P_i' = P_i$
- $(P_i, P_{i+1}) = (NodeRetyping(v,\upsilon,\omega), PreserveNode(v,\omega))$ and $P_i' = P_i$
- $(P_i, P_{i+1}) = (EdgeRetyping(e,v,w,\tau,\phi), PreserveEdge(e,v,w,\phi))$ and $P_i' = P_i$
- $(P_i, P_{i+1}) = (PreserveNode(v\omega), PreserveNode(v,\omega))$ and $P_i' = P_I$
- $(P_i, P_{i+1}) = (PreserveEdge(e,v,w,\tau), PreserveEdge(e,v,w,\tau))$ and $P_i' = P_I$

- $(P_i, P_{i+1}) = (PreserveNode(v,«removed»), Extension(v,\omega))$ and $P_i' = P_{i+1}$
- $(P_i, P_{i+1}) = (PreserveNode(v,\omega), Cancellation(v,\omega))$ and $P_i' = P_{i+1}$
- $(P_i, P_{i+1}) = (PreserveEdge(e,v,w,«removed»), Refinement(e,v,w,\tau))$ and $P_i' = P_{i+1}$
- $(P_i, P_{i+1}) = (PreserveEdge(e,v,w,\tau), Coarsening(e,v,w,\tau))$ and $P_i' = P_{i+1}$
- $(P_i, P_{i+1}) = (PreserveNode(v,\upsilon), NodeRetyping(v,\upsilon,\omega))$ and $P_i' = P_{i+1}$
- $(P_i, P_{i+1}) = (PreserveEdge(e,v,w,\tau), EdgeRetyping(e,v,w,\tau,\phi))$ and $P_i' = P_{i+1}$

Definition 52: Absorbing primitive contract types

In this definition, the case $(NodeRetyping(v,\upsilon_1,\upsilon_2), NodeRetyping(v,\upsilon_2,\upsilon_3))$ leads to an absorption contract type $NodeRetyping(v,\upsilon_1,\upsilon_3)$ when $\upsilon_1 \neq \upsilon_3$ and to an absorption contract type $PreserveNode(v,\upsilon_1)$ if $\upsilon_1 = \upsilon_3$. A similar reasoning leads to the condition $\tau_1 \neq \tau_3$ for $(EdgeRetyping(e,v_1,v_2,\tau_1,\tau_2), EdgeRetyping(e,v_1,v_2,\tau_2,\tau_3))$.

The last 14 cases in the above definition correspond to a special kind of absorption, in the sense that the effect of one of both primitive contract types is completely absorbed in the other one.

Again, all the cases enumerated in the definition above correspond to pairs of sequentially *dependent* contract types. Moreover, in all cases the absorption contract type P_i' **coincides** with the composite contract type $P_i;P_{i+1}$. This leads us to the following property, which is an extension of Property 20.

Replacing *absorbing* pairs of primitive contract types (P_i, P_{i+1}) by their absorption contract type $P_i' = P_i;P_{i+1}$ in a composite reuse contract $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ leads to a reduced composite reuse contract $G_0 \Rightarrow_{P1} \ldots \Rightarrow_{Pi-1} G_{i-1} \Rightarrow_{Pi'} G_{i+1} \Rightarrow_{Pi+2} \ldots \Rightarrow_{Pn} G_n$
Moreover, the reduced reuse contract is unique (after having replaced all absorbing contract types), and has exactly the same final evolution conflicts.

Property 21: Replacing absorbing primitive contract types

Proof:

To prove this property, we only need to show that, in all cases of Definition 52, $P_i' = P_i;P_{i+1}$. Since replacing a sequence of contract types by a composite contract type does not essentially change anything to the composite reuse contract, and because of the confluency property, all other results immediately follow.

- If $(P_i, P_{i+1}) = (NodeRetyping(v,\upsilon,\omega), Cancellation(v,\omega))$ then $PreCond(P_i)=\{(v,\upsilon)\in L_i\}$ and $PostCond(P_i)=\{(v,\omega)\in R_i\}$. Similarly, $PreCond(P_{i+1})=\{(v,\omega)\in L_{i+1}, Adjacent_{Li+1}(v)=\emptyset\}$ and $PostCond(P_{i+1})=\{v\notin R_{i+1}\}$. Applying both contract types sequentially leads to a new contract type $P_i' = P_i;P_{i+1}$ with $PreCond(P_i')=\{(v,\upsilon)\in L'_i, Adjacent_{L'i}(v)=\emptyset\}$ and $PostCond(P_i')=\{v\notin R'_i\}$. Consequently, $P_i'=Cancellation(v,\upsilon)$.
- If $(P_i, P_{i+1}) = (Refinement(e,v,w,\tau), EdgeRetyping(e,v,w,\tau,\phi))$ then $PreCond(P_i)=\{v\in L_i, w\in L_i, (e,v,w)\notin L_i\}$ and $PostCond(P_i)=\{v\in R_i, w\in R_i, (e,v,w,\tau)\in R_i\}$. Similarly, $PreCond(P_{i+1}) = \{v\in L_{i+1}, w\in L_{i+1}, (e,v,w,\tau)\in L_{i+1}\}$ and $PostCond(P_{i+1}) = \{v\in R_{i+1}, w\in R_{i+1}, (e,v,w,\phi)\in R_{i+1}\}$. Applying both contract types sequentially leads to a new contract type $P_i' = P_i;P_{i+1}$ with $ApplCond(P_i')=\{v\in L'_i, w\in L'_i, (e,v,w)\notin L'_i\}$ and $PostCond(P_i')=\{v\in R'_i, w\in R'_i, (e,v,w,\phi)\in R'_i\}$. Consequently, $P_i'=Refinement(e,v,w,\phi)$.
- A similar reasoning can be made for all other cases.

Because the result graph G_n remains the same, and because each contract type P_i that modifies a node or edge in the original sequence has a corresponding contract type P_i' that does the same in the reduced sequence, it is clear that the reduced reuse contract has the same final evolution conflicts as the original one.

V 2.4.5 Minimal Composite Contract Types

In the special case where we have a composite node retyping (resp. composite edge retyping), we can replace the absorbing primitive contract types in such a way that the resulting composite contract type has a very simple form: for any node (resp. edge) in the initial graph there is at most one *NodeRetyping* (resp. *EdgeRetyping*).

A composite node retyping $C_{NodeRet} = P_1{}^*;P_2{}^*;\ldots;P_n{}^*$ is **minimal** if $\forall i\neq j$: $P_i=NodeRetyping(v,\upsilon_1,\upsilon_2)$ and $P_j=NodeRetyping(w,\omega_1,\omega_2)$ implies $v\neq w$.
A composite edge retyping $C_{EdgeRet} = P_1{}^*;P_2{}^*;\ldots;P_n{}^*$ is **minimal** if $\forall i\neq j$: $P_i=EdgeRetyping(e,v_1,v_2,\tau_1,\tau_2)$ and $P_j=EdgeRetyping(f,w_1,w_2,\phi_1,\phi_2)$ implies $(e,v_1,v_2)\neq(f,w_1,w_2)$.
A composite node preservation $C_{NodePres}$ is **minimal** if $\forall i\neq j$: $P_i=PreserveNode(v,\upsilon)$ and $P_j=PreserveNode(w,\omega)$ implies $v\neq w$.
A composite edge preservation $C_{EdgePres}$ is **minimal** if $\forall i\neq j$: $P_i=PreserveEdge(e,v_1,v_2,\tau)$ and $P_j=PreserveEdge(f,w_1,w_2,\phi)$ implies $(e,v_1,v_2)\neq(f,w_1,w_2)$.

Definition 53: Minimal node and edge retyping

Each composite reuse contract $G \Rightarrow_{CNodeRet} H$ (where $C_{NodeRet}$ is a composite node retyping) can be reduced to $G \Rightarrow_{CNodePres} H_1 \Rightarrow_{MNodeRet} H$ where $C_{NodePres}$ is a minimal node preservation and $M_{NodeRet}$ is a minimal node retyping.
Each composite reuse contract $G \Rightarrow_{CEdgeRet} H$ (where $C_{EdgeRet}$ is a composite edge retyping) can be reduced to $G \Rightarrow_{CEdgePres} H_1 \Rightarrow_{MEdgeRet} H$ where $C_{EdgePres}$ is a minimal edge preservation and $M_{EdgeRet}$ is a minimal edge retyping.

Property 22: Minimising composite node and edge retypings

Proof:

We will only show the proof for a composite node retyping $G \Rightarrow_{CNodeRet} H$.
Since *NodeRetypings* involving different nodes are sequentially independent, they can be put in an arbitrary order. This allows us to sort all *NodeRetypings*, given an order on the nodes in the initial graph, e.g., alphabetic order of the unique node labels. Different *NodeRetypings* corresponding to the same node label are left in the order in which they appear in $C_{NodeRet}$.
After sorting, all *NodeRetypings* involving the same node will appear together. Moreover, any two subsequent *NodeRetypings* P_i and P_{i+1} involving the same node v will always have the form $P_i=NodeRetyping(v,\omega,\upsilon)$ and $P_{i+1}=NodeRetyping(v,\upsilon,\omega_2)$. Otherwise, P_{i+1} would not be applicable after P_i, which is impossible since $C_{NodeRet}$ is well-formed.
According to Property 21, (P_i,P_{i+1}) can be replaced by $P_i'=PreserveNode(v,\omega)$ if $\omega=\omega_2$, and by $P_i'=NodeRetyping(v,\upsilon,\omega)$ if $\omega\neq\omega_2$. This process can be repeated until at most one *NodeRetyping* remains for each node involved. In later phases, we also have to take into account situations where $P_i=NodeRetyping(v,\upsilon,\omega)$ and $P_{i+1}=PreserveNode(v,\omega)$ or vice versa. In these cases, the node preservation can be absorbed in a node retyping.
As a final step, all remaining node preservations that have not been absorbed can be moved to the front of the composite reuse contract.

As a corollary of Definition 53, we are able to extend Property 18 for *minimal* composite node retypings, edge retypings, node preservations and edge preservations too.

If $G_0 \Rightarrow_{CComp} G_n$ is a minimal composite node retyping, edge retyping, node preservation or edge preservation, then
(i) $G_0 \Rightarrow_{CComp} G_n$ is a sequentially independent conditional derivation sequence.
(ii) The order of applying the primitive contract types P_i in the composite contract type C_{Comp} is arbitrary, and always leads to the same result.
(iii) There are no applicability conflicts between any two primitive contract types P_i and P_j of C_{Comp}.
(iv) Each ordering leads to the same final evolution conflicts.

Corollary 3: Addendum to Property 18

Intuitively, a composite contract type is **minimal** if it contains the smallest set of primitive contract types needed to obtain the result graph from the initial graph. We already saw an example of this for composite retypings in Definition 53. This definition can be generalised by stating that an arbitrary composite reuse contract is *minimal* if it does not contain any (not necessarily subsequent) absorbing contract types in its sequence.

A composite reuse contract $G_0 \Rightarrow_{P1} G_1 \Rightarrow_{P2} \ldots \Rightarrow_{Pn} G_n$ is **minimal** if
$\nexists\, i \neq j$ such that (P_i, P_j) is absorbing.

Definition 54: Minimal composite reuse contract

V 2.4.6 Normalisation Algorithm

Using the results above, we can transform each composite reuse contract into a minimal one consisting of a composite extension, followed by a composite node retyping, a composite refinement, a composite edge retyping, a composite coarsening, a composite cancellation, a composite node preservation, and finally a composite edge preservation. The process to obtain such a minimal composite reuse contract is called **normalisation**, and the reduced composite reuse contract is called a **normal form**. The normalisation process consists of absorbing pairs of primitive contract types whenever possible, and commuting sequentially independent contract types to place them in the given order. Moreover, the resulting normal form leads to the same final evolution conflicts as the original composite reuse contract.

Each composite reuse contract $G \Rightarrow_{CComp} H$ can be reduced to a minimal sequence of six monotonous composite reuse contracts:

$$G \Rightarrow_{CExt} H_1 \Rightarrow_{CNodeRet} H_2 \Rightarrow_{CRef} H_3 \Rightarrow_{CEdgeRet} H_4 \Rightarrow_{CCoars} H_5 \Rightarrow_{CCanc} H_6 \Rightarrow_{CNodePres} H_7 \Rightarrow_{CEdgePres} H$$

Moreover, the reduced reuse contract has exactly the same final evolution conflicts as $G \Rightarrow_{CComp} H$.

Property 23: Normalisation of a composite reuse contract

Normalisation algorithm

First we give a sketch of the algorithm, and then we discuss in more detail why this algorithm is correct. The idea is reminiscent of a bubble sort algorithm. Whenever two primitive contract types are sequentially independent, they can be commuted. If not, they can be absorbed into a different contract type. By applying this repeatedly, we can bring all primitive contract types of a particular kind to the front of the sequence.

(1) Compare each *Extension* contract type with its predecessor. Commute them if they are sequentially independent. Replace them by an absorbing contract type if they are sequentially dependent. Repeat this until all *Extensions* appear to the front of the sequence.

(2) In the remaining sequence without *Extensions*, repeat the same process for *NodeRetypings*, and minimise the resulting composite node retyping which appears in front of the sequence.

(3) In the remainder, repeat the same process for *Refinements*.

(4) In the remainder, repeat the same process for *EdgeRetypings*.

(5) In the remainder, repeat the same process for *Coarsenings*.

(6) In the remainder, repeat the same process for *Cancellations*. After this, bring all node preservations to the front. As a result, all edge preservations will appear at the end.

(7) When considering the complete resulting sequence, there can still be some absorbing contract types. Remove all absorbing contract types of the form *(Extension, NodeRetyping)*, *(Refinement, EdgeRetyping)*, *(EdgeRetyping, Coarsening)*.

(8) In the new sequence that arises after step (7), there can still occur some redundancies. Remove them in the following order. First remove all absorbing pairs of the form *(Refinement, Coarsening)*. Next, remove redundancies of the form *(Extension, Cancellation)*. Next, consider pairs *(NodeRetyping, Cancellation)*. Finally, remove absorbing pairs of the form *(Extension, PreserveNode)* and *(Refinement, PreserveEdge)*.

Proof:

The proof is an elaboration of the above *algorithm*. It explains in detail the different steps that need to be taken in order to transform a given composite reuse contract into the desired result. *Readers who are not interested in the technical details may skip this proof.*

(1) Number all the contract types in C_{Comp} from P_1 to P_n. Take an arbitrary $P_{i+1} = Extension(v,\omega)$ with $i \geq 1$. If $P_i = Cancellation(v,\upsilon)$ then (P_i,P_{i+1}) can be absorbed by $PreserveNode(v,\upsilon)$ if $\upsilon=\omega$, and by $NodeRetyping(v,\upsilon,\omega)$ if $\upsilon\neq\omega$. If $P_i = Extension(w,\upsilon)$ then do nothing. If $P_i = PreserveNode(v,«removed»)$, simply remove it. In any other case, P_{i+1} and P_i are sequentially independent and can be commuted. Repeat this step until all *Extensions* appear in the front of the sequence, resulting in a composite extension $G \Rightarrow_{CExt} H_1$. In the following steps, only consider the remainder $H_1 \Rightarrow^+ H$, and renumber the remaining contract types from P_1 to P_n.

(2) Take an arbitrary $P_{i+1} = NodeRetyping(v,\upsilon,\upsilon_2)$ with $i \geq 1$. Because all *Extensions* have been filtered out in step **(1)**, $P_i \neq Extension(v,\upsilon)$. If $P_i = PreserveNode(v,\upsilon)$, simply remove it. If $P_i = NodeRetyping(w,\omega,\omega_2)$ (with possibly $v=w$ and $\upsilon=\omega_2$) then do nothing. In any other case, P_{i+1} and P_i are sequentially independent and can be commuted. Repeat this step until all node retypings appear in the front of the sequence, resulting in a composite node retyping $H_1 \Rightarrow_{CNodeRet} H_2$. Minimise this composite node retyping using Property 22. In the following steps, only consider the remainder $H_2 \Rightarrow^+ H$, and renumber the remaining contract types from P_1 to P_n.

(3) Take an arbitrary $P_{i+1} = Refinement(e,v_1,v_2,\tau)$ with $i \geq 1$. Because all *Extensions* and node retypings have been filtered out in steps **(1)** and **(2)**, $P_i \neq Extension(v,\upsilon)$ and $P_i \neq NodeRetyping(v,\upsilon,\omega)$. If $P_i = Coarsening(e,v_1,v_2,\phi)$ then (P_i, P_{i+1}) can be absorbed by $PreserveEdge(e,v_1,v_2,\tau)$ if $\tau=\phi$, and by $EdgeRetyping(e,v_1,v_2,\tau,\phi)$ if $\tau\neq\phi$. If $P_i = Refinement(f,w_1,w_2,\phi)$ then do nothing. If $P_i = PreserveEdge(e,v,w,«removed»)$, simply remove it. In any other case, P_{i+1} and P_i are sequentially independent and can be commuted. Repeat this step until all *Refinements* appear in the front of the sequence, resulting in a composite refinement $H_2 \Rightarrow_{CRef} H_3$. In the following steps, only consider the remainder $H_3 \Rightarrow^+ H$, and renumber the remaining contract types from P_1 to P_n.

(4) Take an arbitrary $P_{i+1} = EdgeRetyping(e,v_1,v_2,\tau,\tau_2)$ with $i \geq 1$. From the previous steps we know that $P_i \neq Extension(v,\upsilon)$, $P_i \neq NodeRetyping(v,\upsilon,\upsilon_2)$ and $P_i \neq Refinement(f,w_1,w_2,\phi)$. If $P_i = PreserveEdge(e,v_1,v_2,\tau)$, simply remove it. If $P_i = EdgeRetyping(f,w_1,w_2,\phi,\phi_2)$ (with possibly $(e,v_1,v_2,\tau)=(f,w_1,w_2,\phi_2)$) then do nothing. In any other case, P_{i+1} and P_i are sequentially independent and can be commuted. Repeat this step until all *EdgeRetypings* appear in the front of the sequence, resulting in a composite edge retyping $H_3 \Rightarrow_{CEdgeRet} H_4$. Minimise this composite edge retyping using Property 22. In the following steps, only consider the remainder $H_4 \Rightarrow^+ H$, and renumber the remaining contract types from P_1 to P_n.

(5) Take an arbitrary $P_{i+1} = Coarsening(e,v_1,v_2,\tau)$ with $i \geq 1$. From the previous steps we know that $P_i \neq Extension(v,\upsilon)$, $P_i \neq NodeRetyping(v,\upsilon,\upsilon_2)$, $P_i \neq Refinement(f,w_1,w_2,\phi)$ and $P_i \neq EdgeRetyping(f,w_1,w_2,\phi,\phi_2)$. If $P_i = PreserveEdge(e,v_1,v_2,\tau)$, simply remove it. All remaining possibilities for P_i are sequentially independent with P_{i+1}, so P_{i+1} and P_i may be commuted. Repeat this step until all *Coarsenings* appear in the front of the sequence, resulting in a composite coarsening $H_4 \Rightarrow_{CCoars} H_5$.

(6) The only remaining primitive contract types are *Cancellations*, node preservations and edge preservations. For each $Cancellation(v,\omega)$ that has a preceding $PreserveNode(v,\omega)$, the latter can be removed. All remaining node and edge preservations can be moved to the end of the sequence, leading to $H_5 \Rightarrow_{CCanc} H_6 \Rightarrow_{CNodePres} H_7 \Rightarrow_{CEdgePres} H$ (a composite cancellation, composite node preservation and composite edge preservation). The result is H, since the only operations that have been performed from step **(1)** to **(6)** are absorption of primitive contract types and commutation of sequentially independent contract types. Indeed, both operations preserve the result graph, according to Property 15 and Property 21.

(7) Consider $G \Rightarrow_{CExt} H_1 \Rightarrow_{CNodeRet} H_2 \Rightarrow_{CRef} H_3 \Rightarrow_{CEdgeRet} H_4 \Rightarrow_{CCoars} H_5 \Rightarrow_{CCanc} H_6 \Rightarrow_{CNodePres} H_7 \Rightarrow_{CEdgePres} H$. We will now remove remaining node and edge retypings that can be absorbed by a different contract type.

- If $\exists P_i = Extension(v,\upsilon)$ in C_{Ext} and $\exists P_j = NodeRetyping(v,\upsilon,\omega)$ in $C_{NodeRet}$, then move P_i to the end of C_{Ext} and move P_j to the beginning of $C_{NodeRet}$. This can be done without any problem because of Property 18 and Corollary 3. After this moving, (P_i,P_j) can be absorbed into $Extension(v,\omega)$.
- If $\exists P_i = Refinement(e,v,w,\tau)$ in C_{Ref} and $\exists P_j = EdgeRetyping(e,v,w,\tau,\phi)$ in $C_{EdgeRet}$, then move P_i to the end of C_{Ref} and move P_j to the beginning of $C_{EdgeRet}$. This can be done without any problem because of Property 18 and Corollary 3. After this moving, (P_i,P_j) can be absorbed into $Refinement(e,v,w,\phi)$.

- If $\exists P_i$ = *EdgeRetyping(e,v,w,τ,ϕ)* in $C_{EdgeRet}$ and $\exists P_j$ = *Coarsening(e,v,w,ϕ)* in C_{Coars}, then move P_i to the end of $C_{EdgeRet}$ and move P_j to the beginning of C_{Coars}. This can be done without any problem because of Property 18 and Corollary 3. After this moving, *(P_i,P_j)* can be absorbed into *Coarsening(e,v,w,τ)*.
- The case of *NodeRetypings* that are absorbed by a node *Cancellation* will be postponed to step **(8)**.
- Repeat this step until all superfluous *NodeRetypings* and *EdgeRetypings* have been removed. As result of this step, the sequence will only contain *NodeRetypings* and *EdgeRetypings* that do not correspond to nodes or edges occurring in an *Extension*, *Refinement* or *Coarsening*.

(8) Consider the remaining sequence $G \Rightarrow_{CExt} H'_1 \Rightarrow_{CNodeRet} H'_2 \Rightarrow_{CRef} H'_3 \Rightarrow_{CEdgeRet} H'_4 \Rightarrow_{CCoars} H'_5 \Rightarrow_{CCanc} H_6 \Rightarrow_{CNodePres} H_7 \Rightarrow_{CEdgePres} H$. To obtain a minimal composite reuse contract, we still need to remove all remaining redundant pairs. This needs to be done in a particular order:

- If $\exists P_i$ = *Refinement(e,v,w,τ)* in C_{Ref} and $\exists P_j$ = *Coarsening(e,v,w,τ)* in C_{Coars}, then move P_i to the end of C_{Ref} and move P_j to the beginning of C_{Coars}. This can be done without any problem because of Property 18. We can even move P_i to the end of $C_{EdgeRet}$ since all absorbing situations have been removed in step **(7)**. As a result, *(P_i,P_j)* becomes a redundant pair that can be removed. Repeat this step until no redundant *Refinements* and *Coarsenings* remain.
- If $\exists P_i$ = *Extension(v,ω)* in C_{Ext} and $\exists P_j$ = *Cancellation(v,ω)* in C_{Canc}, then move P_i to the end of C_{Ext} and move P_j to the beginning of C_{Canc}. This can be done without any problem because of Property 18. We can even move P_i to the end of $C_{NodeRet}$ because all absorbing situations have been removed in step **(7)**. Because $\{Adjacent_{Ri}(v)=\emptyset\} \subseteq PostCond(P_i)$ and $\{Adjacent_{Lj}(v)=\emptyset\} \subseteq PreCond(P_j)$, any edge to or from *v* occurring in an intermediary graph must have been introduced by a *Refinement* after the *Extension*, and must have been removed again by a *Coarsening* before the *Cancellation*. However, since all redundant *Refinements* and *Coarsenings* have been removed in the first part of step **(8)**, we can safely conclude that there are no remaining *Refinements* or *Coarsenings* referring to *v*. Hence, we can move P_i to the end of C_{Ref} and P_j to the beginning of C_{Coars}, because P_i is sequentially independent of the remaining *Refinements*, and P_j is sequentially independent of the remaining *Coarsenings*. We can even move P_j to the beginning of $C_{EdgeRet}$, since all absorbing situations have been removed in step **(7)**. Once this is done, *(P_i,P_j)* is redundant and can be removed. Repeat this step until all such redundant pairs have been removed.
- As a next step, if $\exists P_i$ = *NodeRetyping(v,υ,ω)* in $C_{NodeRet}$ and $\exists P_j$ = *Cancellation(v,ω)* in C_{Canc}, then P_i and P_j can be absorbed into P_i' = *Cancellation(v,υ)*.
- Finally, if $\exists P_i$ = *Extension(v,υ)* and $\exists P_j$ = *PreserveNode(v,υ)*, or $\exists P_i$ = *Cancellation(v,υ)* and $\exists P_j$ = *PreserveNode(v,«removed»)*, or $\exists P_i$ = *Refinement(e,v,w,τ)* and $\exists P_j$ = *PreserveEdge(e,v,w,τ)*, or $\exists P_i$ = *Coarsening(e,v,w,τ)* and $\exists P_j$ = *PreserveEdge(e,v,w,«removed»)* then P_j can be removed.

By construction, the resulting composite reuse contract is minimal, since all absorbing primitive contract types have been reduced. Moreover, the result *H* remains the same because we have only commuted sequentially independent contract types and removed absorbing contract types. Consequently, the final evolution conflicts remain the same as they are detected by looking at the result graph only.

Although the normalisation algorithm always leads to a *minimal* composite reuse contract, the constructed normal form is *not unique*. Alternatively, each composite reuse contract can be reduced to $G \Rightarrow_{CNodePres} H_1 \Rightarrow_{CEdgePres} H_2 \Rightarrow_{CNodeRet} H_3 \Rightarrow_{CEdgeRet} H_4 \Rightarrow_{CCoars} H_5 \Rightarrow_{CCanc} H_6 \Rightarrow_{CExt} H_7 \Rightarrow_{CRef} H$. There are even many other possibilities. Nevertheless, all these minimal composite reuse contracts have something in common. They all contain the same primitive contract types, but possibly in a different order. This leads us to the following important result:

> The normal form of a composite reuse contract $G \Rightarrow_{CComp} H$ is unique modulo a permutation of its primitive contract types.

Property 24: Uniqueness of the normal form

Proof:

Suppose that $\exists$ two minimal reductions $G \Rightarrow_{M1} H$ and $G \Rightarrow_{M2} H$ of $G \Rightarrow_{CComp} H$. Then they can only differ in the order of their contract types.

We will prove that $\forall$ primitive contract type $P \in M_1$: $P \in M_2$

(1) If P = *Extension*$(v,\upsilon) \in M_1$ then P cannot be followed (directly or indirectly) by a *Cancellation*(v,υ) since redundant pairs have been removed. Likewise, P cannot be followed by a *Cancellation*(v,ω). Indeed, the preconditions of the latter can only be satisfied if, after the *Extension* P, a *NodeRetyping*(v,υ,ω) has occurred. But because M_1 is minimal, all *NodeRetypings* of v have already been absorbed in P. Due to the orthogonality of the primitive contract types, *Cancellation* is the only possible way to remove nodes, so we can conclude that v is newly introduced by M_1, i.e., $v \notin G$ but $(v,\upsilon) \in H$. Because M_2 has the same initial and result graph as M_1, it must also contain the same *Extension*(v,υ). Indeed, *Extension* is the only way to introduce nodes. Moreover, there can be no *Extension*(v,ω) in M_2, since $(v,\upsilon) \in H$ and M_2 is minimal, and hence does not contain any *NodeRetypings* of v.

(2) If P = *Refinement*$(e,v,w,\tau) \in M_1$, then the minimality of M_1 guarantees that there are no *EdgeRetypings* of (e,v,w), since they have all been absorbed due to the presence of P. Minimality also guarantees that there is no *Coarsening*(e,v,w,τ) since it would be redundant with P. Consequently, the presence of P guarantees that $(e,v,w) \notin G$ but $(e,v,w,\tau) \in H$. Moreover, due to the orthogonality of the primitive contract types, *Refinement* is the only way to introduce a new edge, so any other minimal reduction M_2 that adds an edge (e,v,w,τ) to a graph can only achieve this by means of the same *Refinement* P. Hence, $P \in M_2$.

(3) If P = *NodeRetyping*$(v,\upsilon,\omega) \in M_1$ then the minimality of M_1 guarantees that P is the only *NodeRetyping* of v, and there are no *Extensions* or *Cancellations* of v (since these would have absorbed the *NodeRetyping*). Consequently, v is not newly reduced or removed by M_1. Only the type of v is changed: $(v,\upsilon) \in G$ and $(v,\omega) \in H$. In M_2, such a change of the node type could also have been achieved in a different way: by first removing v and then reintroducing v. However, this requires a *Cancellation* of v followed by an *Extension* of v with a new type, and all such situations are absorbed in a *NodeRetyping* of v. Hence, the minimality of M_2 also requires the presence of P in order to achieve a retyping of v.

(4) If P = *PreserveNode*$(v,\upsilon) \in M_1$, then the minimality of M_1 guarantees that P is the only node preservation of v, and there are no *Extensions*, *Cancellations* or *NodeRetypings* of v. Consequently, v is unaltered by M_1. Nevertheless, the fact that a node preservation of v is present in M_1 can be seen in the result graph H, where (v,υ) will have a different modification tag than in G. Because M_2 is also minimal, the only modification which allows a node (v,υ) to be present in both G and H, while its modification tag has been changed, is by means of the same node preservation P.

(5) The proof for *Cancellation*, *Coarsening*, *EdgeRetyping* and *PreserveEdge* is analogous.

Note that, in the above property, the reorderings are not completely arbitrary. Sometimes, an *Extension* must be performed first before a *Refinement* can take place. This is the case if the *Refinement* adds an edge between nodes of which at least one was not present in the original graph G. Vice versa, sometimes a *Coarsening* must happen before a *Cancellation* can be made. This is the case if a node needs to be removed that still contains adjacent edges in the original graph G. Except for these constraints, the primitive contract types in the normal form can be put in an arbitrary order.

V 2.4.7 Discussion

We already mentioned the most obvious advantage of normalisation before. It makes the evolution sequence more comprehensible, by removing all redundant information, and by grouping the same kind of modifications together in a monotonous contract type. In this way, the evolution sequence will become shorter, which will also make the conflict detection algorithm more efficient. Because there are fewer contract types, there are fewer cases in which conflicts need to be detected. Additionally, because normalisation removes redundant contract types, intermediate redundant conflicts are removed as well.

A normalised reuse contract also has the important advantage that it is much easier to find out the precise changes that have been made between two versions of the software. For example, suppose we want to find all the classes (or methods) that have been added to an object-oriented framework by a sequence of evolution steps. Since all of these additions correspond to *Extensions* of nodes with type *«class»*, they will be clustered in the resulting normal form. Hence, such a question can be answered very efficiently, and doesn't have the disadvantage that it includes items that are irrelevant (because they are removed later on). Similarly, if we want to find only one specific change that has been performed during subsequent evolutions, this can be achieved in a very efficient way.

When comparing two subsequent versions of a software application, where the modifications have been documented with reuse contracts, inspection of the redundant parts sometimes allows us to find remaining bugs or inconsistencies in the new version. For example, suppose that during software evolution one bug fix has been solved by adding a particular edge. Then it is possible that this modification introduces a new conflict somewhere else. In order to solve this new conflict, a second bug fix is made, removing the first edge, and thus reintroducing the first bug. This process can go on for quite a while before being detected, especially when different persons (each responsible for different parts of the software) are involved in making the bug fixes. The only way to solve the problem is by solving both bug fixes simultaneously. From a formal point of view, the problem can be recognised as a recurring sequence of redundant pairs (or, more generally, tuples). First a *Refinement* introduces an edge, then a *Coarsening* removes the edge, then a *Refinement* reintroduces the edge, etc… It this occurs repeatedly, there is probably a problem. Note that this problem can only be *detected before normalisation*, since all redundant pairs are removed during the normalisation process.

Another problem that can only be detected before normalisation has to do *with accidental name collisions*. This problem only occurs when items are removed somewhere in the evolution sequence, and reintroduced later on. It occurs when a $\mathit{Coarsening}(e,v,w,\tau)$ is followed by a $\mathit{Refinement}(e,v,w,\phi)$, or when a $\mathit{Cancellation}(v,\upsilon)$ is followed after a while by an $\mathit{Extension}(v,\omega)$. The normalisation algorithm will reduce the latter case to a single $\mathit{NodeRetyping}(v,\upsilon,\omega)$ if $\upsilon\neq\omega$, or to $\mathit{PreserveNode}(v,\upsilon)$ if $\upsilon=\omega$. This is not a good solution if the node *v* introduced by the $\mathit{Extension}(v,\omega)$ has nothing in common with the original node *v* that was cancelled before. We say that the label *v* of the new node gives rise to an *accidental name collision*. This can lead to a conflict if an independent reuser makes a change to the original component that used the old version of *v*, for example by adding an edge to this node. If we want to integrate the changes of this reuser in the new version, we cannot simply add an edge to the new version of *v*, because *v* might have a new meaning. Again, this problem can only be detected *before normalisation*, or by *logging all accidental name collisions* between different versions of the software during the normalisation process.

The opposite cases, where we first perform an $\mathit{Extension}(v,\upsilon)$ and later a $\mathit{Cancellation}(v,\upsilon)$, or first a $\mathit{Refinement}(e,v,w,\tau)$ and later a $\mathit{Coarsening}(e,v,w,\tau)$, do not give rise to accidental name collisions. Both cases can be considered as the use of a *temporary node v* (or *temporary edge (e,v,w)*) that is introduced and removed again during the evolution process.

V 2.4.8 Extraction Algorithm

When only the initial graph and the result graph of a reuse contract are given, it is still possible to extract a minimal composite reuse contract that leads from the initial graph to the result graph. In other words, we can reconstruct the primitive contract types that lead to a particular result graph. The proof of this property is given by means of an algorithm, and a sketch of proof that this algorithm is correct.

> If *H* is obtained from *G* by means of a composite reuse contract $G \Rightarrow_{CComp} H$, then a normalised reuse contract $G \Rightarrow_{CExtract} H$ can be automatically extracted by comparing *G* and *H*.

Property 25: Extraction of a normalised reuse contract

Extraction Algorithm

Given the graphs *G* and *H*, we can calculate composite contract type $C_{Extract}$ by performing the following steps:

(1) For each node *v* such that $v\notin G$ and $(v,\upsilon)\in H$: add $\mathit{Extension}(v,\upsilon)$ to $C_{Extract}$.

(2) For each node *v* such that $(v,\upsilon)\in G$ and $(v,\omega)\in H$: add $\mathit{NodeRetyping}(v,\upsilon,\omega)$ to $C_{Extract}$.

(3) For each edge (e,v,w) such that $(e,v,w)\notin G$ and $(e,v,w,\tau)\in H$: add $\mathit{Refinement}(e,v,w,\tau)$ to $C_{Extract}$.

(4) For each edge (e,v,w) such that $(e,v,w,\tau)\in G$ and $(e,v,w,\phi)\in H$: add $\mathit{EdgeRetyping}(e,v,w,\tau,\phi)$ to $C_{Extract}$.

(5) For each edge (e,v,w) such that $(e,v,w,\tau)\in G$ and $(e,v,w)\notin H$: add $\mathit{Coarsening}(e,v,w,\tau)$ to $C_{Extract}$.

(6) For each node v such that $(v,\upsilon)\in G$ and $v\notin H$: add *Cancellation*(v,υ) to $C_{Extract}$.

(7) For each node v such that $(v,\upsilon)\in G$ and $(v,\upsilon)\in H$ with different modification tags: add *PreserveNode*(v,υ) to $C_{Extract}$.

(8) For each edge (e,v,w) such that $(e,v,w,\tau)\in G$ and $(e,v,w,\tau)\in H$ with different modification tags: add *PreserveEdge*(e,v,w,τ) to $C_{Extract}$.

Sketch of proof:

To check validity of the above extraction algorithm, it suffices to show that the extracted composite reuse contract $G \Rightarrow_{CExtract} H$ is well-formed and minimal. Property 24 then guarantees the uniqueness of this minimal composite reuse contract.

First, we show that $C_{Extract}$ is a *well-formed* composite contract type, in the sense that there are no applicability conditions that are not satisfied in the sequence.

- For any P_i = *Extension*(v,υ), $\{v\notin L_i\} \subseteq$ *PreCond*(P_i) is satisfied because $v\notin G$ and $C_{Extract}$ contains at most one *Extension* for each node v.
- For any P_i = *NodeRetyping*(v,υ,ω), $\{(v,\upsilon)\in L_i\} \subseteq$ *PreCond*(P_i) is satisfied, since $(v,\upsilon)\in G$, $C_{Extract}$ contains at most one *NodeRetyping* for each node v, and the *Extensions* that precede P_i do not introduce the same node v (because all cases are disjoint).
- For any P_i = *Refinement*(e,v,w,τ), $\{v\in L_i,\ w\in L_i,\ (e,v,w)\notin L_i\} \subseteq$ *PreCond*(P_i) is satisfied. $v\in L_i$ and $w\in L_i$ since the nodes v and w were already present in G or have been introduced by the *Extensions*. $(e,v,w)\notin L_i$ since $(e,v,w)\notin G$ by construction, and *Refinements* that precede P_i do not introduce the same edge (because all cases are disjoint).
- For any P_i = *PreserveNode*(v,υ), $\{(v,\upsilon)\in L_i\} \subseteq$ *PreCond*(P_i) is satisfied because $(v,\upsilon)\in G$ and all cases are disjoint.
- In an analogous way we can continue for *Coarsening*, *Cancellation*, *EdgeRetyping* and *PreserveEdge*.

Next, we show that $C_{Extract}$ is a *minimal* composite contract type.

- *Redundant* contract types like (*Extension*(v,υ),*Cancellation*(v,υ)) are impossible by construction, since we require that $v\notin G$ and $(v,\upsilon)\in H$ for *Extension*, and $(v,\upsilon)\in G$ and $v\notin H$ for *Cancellation*. In both cases, either the first or the second condition is not satisfied. The same holds for all other redundant contract types.
- *Absorbing* contract types do not occur. For example, (*Extension*(v,υ), *NodeRetyping*(v,υ,ω)) is impossible, since $\{v\notin L\} \subseteq$ *PreCond*(*Extension*(v,υ)) contradicts with $\{(v,\upsilon)\in L\} \subseteq$ *PreCond*(*NodeRetyping*(v,υ,ω)). In an analogous way, all other absorbing contract types are impossible.

V 2.5 Experiments

In this section we introduced several algorithms which are necessary to provide automated support for evolution. First of all, we explained our *conflict detection algorithm*, which allowed us to detect applicability and evolution conflicts between parallel evolutions of the same software artifact. We also briefly addressed the need to supplement conflict detection with a semi-automated *conflict resolution algorithm*, which provides support to resolve the conflicts once they have been detected. In order to reduce the complexity and increase the understandibility of a composite contract type, and also to make the conflict detection process more efficient, we introduced a *normalisation algorithm*. This allowed us to remove all redundant information from an arbitrary composite contract type. Finally, given two subsequent versions of the same software artifact, it is also possible to extract automatically reuse contract information by means of a so-called *extraction algorithm*.

We will now discuss in more detail the experiments that have been made with these various algorithms. Most of the earlier experiments have been performed in Smalltalk, since the original reuse contracts paper [Steyaert&al96] addressed the need to deal with the fragile base class problem in (Smalltalk) class inheritance hierarchies. To validate the new ideas that are discussed in this dissertation, we developed prototype implementations in Mathematica and PROLOG.

V 2.5.1 Conflict Detection Algorithm

In large software systems, manually checking evolution conflicts is practically unfeasible due to the size of the system as well as the large number of evolution conflicts that are usually detected. Therefore, tool support is an absolute necessity.

A prototype version of the conflict detection algorithm has been implemented in PROLOG. It is implemented in the way described in this dissertation. PROLOG has shown to be a very flexible language for implementing the algorithm, because of its powerful unification mechanism. Obviously, the implementation is made in a domain-independent way, so that it can be customised easily to different application domains.

V 2.5.2 Conflict Resolution Algorithm

A conflict resolution algorithm has not yet been implemented, because this is only possible based on domain-specific information. Depending on the specific domain, other resolution techniques will be needed. Therefore, we first need to customise the domain-independent formalism (and its corresponding implementation) to specific domains before we can address the issue of conflict resolution.

Some preliminary work in this direction has already been performed in [Mezini97], where some of the evolution conflicts mentioned in this dissertation were not only detected, but could even be solved in a semi-automatic way. The approach was dedicated to the domain of object-oriented class hierarchies, and an implementation was made in Smalltalk.

Another interesting approach for solving conflicts when a set of conflicting primitive transformations is detected has been explained in [Lippe&vanOosterom92]. To solve a conflict, there are three alternative strategies:

- Impose a fixed order on the primitive transformations that are being merged. This is for example useful if we have an item the value of which is changed in two different ways, and we want to retain one of these new values in the final merge result.
- Delete some of the primitive transformations. This is necessary if parallel changes add the same behaviour (possible in different ways). To avoid code duplication or performing the same behaviour twice, some of the primitive transformations need to be deleted.
- Edit some of the transformations, or add new ones. This is a last resort if the previous options are not sufficient to solve the problems.

One should be aware that the last two strategies may change the existing conflicts. As a result, it is necessary to (partially) perform the conflict detection algorithm again to find out if new conflicts will be introduced.

V 2.5.3 Normalisation and Extraction Algorithm

The normalisation and extraction algorithms are closely related to each other. An early version of these algorithms was implemented in Smalltalk by Koen De Hondt, to validate the claims made in [Steyaert&al96] in the context of object-oriented class hierarchies. However, because of the immaturity of the reuse contract formalism at that time, the normalisation algorithm was too restrictive to be practical in large experiments.

In [DeHondt98], a new version of the extraction algorithm was implemented in the context of collaborating classes. Again, this algorithm was restricted to a specific domain.

In an attempt to generalise the normalisation and extraction algorithms, and make them domain-independent, a prototype implementation has been developed in *Mathematica*, using the *Combinatorica*-package for dealing with graphs. Based on this experiment, the algorithms that can be found in this dissertation were developed. A new version of both algorithms (normalisation and extraction) will be added to the already existing PROLOG-implementation of the domain-independent reuse contract framework. In an industrial case study that is going on with an industrial partner, we intend to validate these algorithms in practice.

V 2.5.4 Integration in a Tool

In order to be really practically useful, all the algorithms mentioned above should be integrated into a software development environment or CASE tool. Moreover, the software developer should not be aware of the underlying mechanisms of the reuse contract approach. When developing or modifying software, the tool should automatically generate the corresponding reuse contracts. When two separately modified versions of the same piece of software are merged, the conflict detection algorithm could be invoked to find out if there are any undesired interactions. If this is the case, the conflict resolution algorithm can be executed to assist the developer in solving the conflicts. Some conflicts, like name clashes, can be resolved automatically, while others will require more feedback from the software developer.

V 2.5.5 Performance Issues

The prototype implementations in Mathematica and PROLOG did not take any performance issues into account. Especially the normalisation algorithm can be implemented in a much more efficient way, if we decide to fix a certain order of composite contract types. The conflict detection algorithm can also be improved significantly by using more intelligent algorithms and heuristics for detecting graph patterns in a particular graph. Finally, if we want to detect transitive closure conflicts as well, we should make use of more efficient algorithms for computing the transitive closure.

V . 3 PREDEFINED COMPOSITE CONTRACT TYPES

In order to address the scalability of reuse contracts, we introduced composite contract types in the previous section, as well as a normalisation algorithm to remove redundancy in an arbitrary composite evolution sequence. In this section we look at some useful predefined combinations of primitive contract types. Until now, we have only predefined *monotonous* composite contract types that are composed of primitive contract types that all have *the same kind*. For example, a composite extension only contains *Extensions* as primitive contract types.

We will now look at predefined composite contract types that are composed of *different kinds* of primitive contract types. Some of these predefined composite types specify additional constraints that must hold between these primitive contract types. These constraints allow us to provide more specific information about how the primitive constituents of a composite contract type are supposed to work together. This allows us to ignore some of the evolution conflicts that occur with a primitive contract type if it is part of a composite contract type.

Formally, a composite contract type (as defined in Definition 48 of page 110) was nothing more than a sequential composition of a number of (usually sequentially dependent) primitive contract types. Because it is visually more attractive, we will use the notation ***[P_1,…,P_n]*** instead of P_1*;P_2*;…P_n* to denote the sequence of primitive contract types a composite contract type is built of. Obviously, the contract types P_i may also be composite again instead of primitive ones. In this way, arbitrarily complex contract types can be created.

V 3.1 FACTORISATION

V 3.1.1 Example

A first example of a predefined composite contract type is ***Factorisation***, which is an operation that factors out some common behaviour, and puts it in a new component. This allows one to create more generic components, at the expense of introducing an extra level of indirection. An example of a *Factorisation* could already be seen in Figure 23 of page 77, but for convenience we will recapitulate the essential part of this example below.

Suppose that, in an early design phase, we have two geometrical objects *Circle* and *Triangle*, which both refer to a *Point* object via a *«has-a»* relationship called *center*. In a later design phase we try to refactor the common behaviour between both geometrical objects, by introducing a new intermediary object *Geo*. Both direct relationships from *Circle* to *Point* and from *Triangle* to *Point* are replaced by indirect dependencies via the intermediary object *Geo*. This composite modification is graphically represented in Figure 30. It is easy to see that it consists of the following composite contract type:

[Coarsening(center,Circle,Point,«hasa»), Coarsening(center,Triangle,Point,«hasa»),

Extension(Geo,«object»), Refinement(ε,Circle,Geo,«isa»), Refinement(ε,Triangle,Geo,«isa»),

Refinement(center,Geo,Point,«hasa»)].

If desired, the two *Coarsenings* and three *Refinements* may be replaced by a composite coarsening and a composite refinement, respectively. Moreover, the ordering of the contract types in the sequence is irrelevant, as long as the *Extension* is performed before the *Refinements*, because of their sequential dependence.

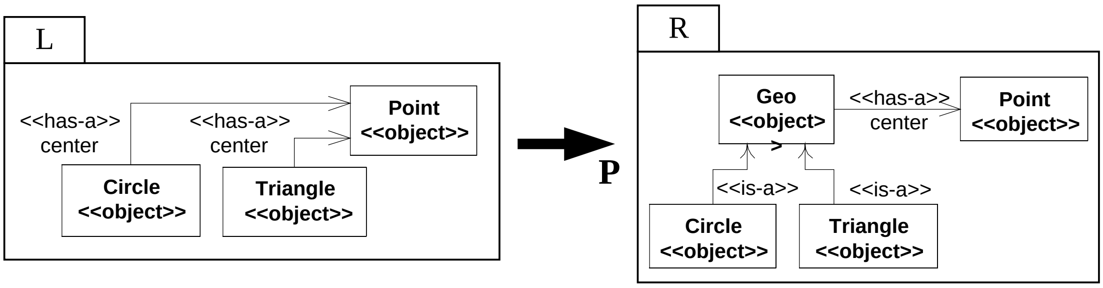


Figure 30: Example of a *Factorisation*

As can be seen in this example, a *Factorisation P: L→R* needs to satisfy one important invariant: **after factorisation, all dependencies that were present in L still need to be present in R, possibly through an extra level of indirection.** In Figure 30, *L* contains an edge from *Circle* to *Point*, and an other one from *Triangle* to *Point*. After the factorisation *P: L→R*, *R* still contains a dependency from *Circle* to *Point*, but indirectly through *Geo*. An analogous reasoning can be made for the dependency from *Triangle* to *Point*. Stated otherwise, the *Factorisation* needs to satisfy the invariant that the transitive closure graph, restricted to the nodes that already existed in *L*, remains the same.

V 3.1.2 Detecting Evolution Conflicts

One of the advantages of predefined composite contract types is that they allow us to fine-tune the detection of evolution conflicts in some cases. For example, the *Factorisation* of Figure 30 would, among others, lead to a *EC1: Reachability conflict* if a different modifier would introduce an extra edge from an other node to *Circle*. Indeed, the *Factorisation* removes an edge from *Circle* to *Point*, with as unforeseen side-effect that nodes that refer to *Circle* can no longer reach *Point*.

However, because of the specific characteristics of *Factorisation*, namely that the transitive closure of edges is preserved, this evolution conflict may be ignored. Indeed, it is still possible for *Circle* to reach *Point*, but indirectly through the intermediate node *Geo*. All other reachability conflicts that arise because of one of the *Refinements* or *Coarsenings* in the *Factorisation* can be ignored in the same way. Note that not all evolution conflicts may be ignored. For example, all conflicts in which *Geo* plays an explicit role still need to be detected.

Actually, the underlying reason for the fact that some evolution conflicts may be ignored for a *Factorisation*, is that it is a kind of behaviour-preserving transformation. In [Opdyke92] and [Bergstein94] some other behaviour-preserving transformations are presented as well, but we will not discuss them here. Each of these behaviour-preserving transformations will allow us to ignore some of the evolution conflicts that will be detected at the lower level of primitive contract types.

To summarise, we can say that the use of predefined composite contract types simplifies the conflict detection process, since **some evolution conflicts need not be detected in the presence of particular predefined composite contract types**. Depending on the intuitive semantics that is associated to a composite contract type, we can ignore some of the more primitive evolution conflicts, because they are not considered important if they occur inside a composite contract type.

V 3.2 OTHER COMPOSITE CONTRACT TYPES

We will now briefly mention other predefined composite contract types that, according to our experiments, have shown to be useful in practice. Obviously, many other interesting composite contract types can be found, but this subsection merely serves to demonstrate the underlying idea.

Note that, for each of the composite contract types mentioned below, we do not explicitly express their application conditions, since these follow immediately from the application conditions of the primitive contract types they are composed of.

V 3.2.1 Redirect Edges

The first three composite contract types describe what happens when a single edge is modified by changing its source and/or target nodes. Three important cases are distinguished: *RedirectEdge*, *RedirectSource* and *RedirectTarget.*

RedirectEdge(e,u,v,τ)

> **Description.** Change the direction of an edge by swapping its source and target nodes. This composite contract type is composed of a *Coarsening* and a *Refinement*.
> **Definition:** *RedirectEdge(e,u,v,τ) = [Coarsening(e,u,v,τ), Refinement(e,v,u,τ)]*

RedirectSource(e,u,v,τ,w)

Description. Redirect the *source* of an edge to a new node that has the same type as the original source node. This composite contract type is composed of a *Coarsening* and a *Refinement*.
Definition: *RedirectSource(e,u,v,τ,w) = [Coarsening(e,u,v,τ), Refinement(e,w,v,τ)]*
with additional precondition that *type(u)=type(w)*.

In Figure 30 we already saw two examples of the above modification. *RedirectSource(center,Circle,Point,«hasa»,Geo)* could be used to change the source of the *center* edge from *Circle* to *Geo*. This allows us to express *Factorisation* in terms of this composite contract type, which is more intuitive than the one shown in section V 3.1.1:

[Extension(Geo,«object»), Refinement(ε,Circle,Geo,«isa»), Refinement(ε,Triangle,Geo,«isa»),

RedirectSource(center,Circle,Point,«hasa»,Geo), Coarsening(center,Triangle,Point,«hasa»)].

As our next predefined composite reuse contract, we define *RedirectTarget* as the equivalent of *RedirectSource*, but for changing the target of an edge.

RedirectTarget(e,u,v,τ,w)

Description. Redirect the *target* of an edge to a new node that has the same type as the original target node. This composite contract type is composed of a *Coarsening* and a *Refinement*.
Definition: *RedirectTarget(e,u,v,τ,w) = [Coarsening(e,u,v,τ), Refinement(e,u,w,τ)]*
with additional precondition that *type(v) = type(w)*.

While the composite contract types *RedirectSource* and *RedirectTarget* only redirect one edge by changing its source or target nodes, we sometimes need an operation to redirect **all** incoming edges or outgoing edges from a node. For this purpose, we introduce the following composite contract types that are defined in terms of *RedirectSource* and *RedirectTarget*.

RedirectOutgoing(u,w)

Description. Redirect *all* edges that have a particular node as *source* to a new source node of the same type.
Definition:
RedirectOutgoing(u,w) = [RedirectSource(e_1*,u,*v_1*,*τ_1*,w),…,RedirectSource(*e_n*,u,*v_n*,*τ_n*,w)]*
with additional restrictions that *type(u) = type(w)* and *OutEdge*$_G$*(u) = {(*e_1*,u,*v_1*)…,(*e_n*,u,*v_n*)}* (as defined in Definition 7 of page 45)

RedirectIncoming(v,w)

Description. Redirect *all* edges that have a particular node as *target* to a new target node of the same type.
Definition:
RedirectIncoming(v,w) = [RedirectTarget(e_1*,*u_1*,v,*τ_1*,w),…,RedirectTarget(*e_n*,*u_n*,v,*τ_n*,w)]*
with additional restrictions that *type(v) = type(w)* and *InEdge*$_G$*(v) = {(*e_1*,*u_1*,v)…,(*e_n*,*u_n*,v)}* (as defined in Definition 7 of page 45)

Unlike *Factorisation*, the composite contract types presented above do not reduce the number of evolution conflicts.

V 3.2.2 Connected Extensions and Extending Refinements

In [Lucas97], the following two composite contract types were equally defined.

ConnectedExtension

> **Description.** Add a new node *u* to the graph, as well as edges *from this node to* existing nodes in the graph. All these added edges must have the same type τ. This composite contract type is composed of an *Extension* and an arbitrary number of *Refinements*. Alternatively, all these *Refinements* can be bundled in a composite refinement, with as additional constraint that all its constituent *Refinements* must introduce edges with the same type τ and source node *u*.
>
> **Definition:** $ConnectedExtension(u,\omega,\tau,[(e_1,v_1),\ldots,(e_n,v_n)]) =$
> $[Extension(u,\omega), Refinement(e_1,u,v_1,\tau), \ldots, Refinement(e_n,u,v_n,\tau)]$

ExtendingRefinement

> **Description.** Add a new node *v* to the graph, as well as edges *to this node from* existing nodes in the graph. All these added edges must have the same type τ. An extending refinement is composed of an *Extension* followed by an arbitrary number of *Refinements*. Alternatively, all these *Refinements* can be bundled in a composite refinement, with as additional constraint that all its constituent *Refinements* must introduce edges with the same type τ and target node *v*.
>
> **Definition:** $ExtendingRefinement(v,\omega,\tau,[(e_1,u_1),\ldots,(e_n,u_n)]) =$
> $[Extension(v,\omega), Refinement(e_1,u_1,v,\tau),\ldots,Refinement(e_n,u_n,v,\tau)]$

Using this *ExtendingRefinement* we can simplify the composite contract type *Factorisation* of section V 3.1.1 even further to:

[ExtendingRefinement(Geo,«object»,«isa»,[(ε,Circle),(ε,Triangle)]),

RedirectSource(center,Circle,Point,«hasa»,Geo), Coarsening(center,Triangle,Point,«hasa»)].

V 3.2.3 Merging Nodes

As a final predefined composite contract type, we define *MergeNodes*, which unifies an arbitrary number of nodes into a single new node.

MergeNodes

> **Description.** Merge an arbitrary number of existing nodes into a single new node. The targets of the incoming edges of all the merged nodes are redirected to the new node, and the sources of the outgoing edges of all the merged nodes are redirected to the new node as well. Obviously, this redirection should not introduce applicability conflicts such as *AC5: Duplicate edge conflict*.
>
> **Definition:** $MergeNodes([u_1,u_2,\ldots,u_n],v) = [RedirectIncoming(u_1,v), RedirectOutgoing(u_1,v), \ldots, RedirectIncoming(u_n,v), RedirectOutgoing(u_n,v)]$ with the additional restriction that the redirection does not cause any applicability conflicts.

V . 4 NESTING

Until now we have not considered nested graphs in our formal treatment of reuse contracts because we wanted to avoid the additional technical difficulties they give rise to for as long as possible. Nevertheless, nesting is essential to deal with the inherent complexity of large graphs, since it allows us to encapsulate nodes inside other ones, and thus provides a kind of layering and encapsulation mechanism.

This section discusses the changes that need to be made to the reuse contract framework in order to deal with nested graphs. More precisely, the primitive contract types *Extension* and *Cancellation* need to be modified slightly, as well as the possible evolution conflicts. We also introduce some other useful primitive contract types in the presence of nesting.

V 4.1 NOTATIONAL CONVENTIONS

Note to the reader.

> *Before continuing, it is useful to go back to section III 2.3 on page 51 and review the formal definition of nested graphs.*

Essentially, a nested graph is a labelled typed graph with an extra relation *nested* defined on the nodes of the graph. This relation defines a nesting hierarchy. If nodes are nested in other nodes, they are represented visually by drawing them inside each other.

Nodes that are not directly nested in the same parent node may have the same label. To uniquely refer to a nested node, we take the convention of qualifying its node label by the label of its parent nodes. For example, we write *A.B* if a node *w* with label *B* is nested in a node *v* with label *A*, i.e., $(w,v) \in nested$.

A rather technical problem arises because there are essentially two different kinds of nodes: top-level nodes and nodes that are nested in a different one. While the labels of the latter need to be qualified by their parent's labels, this is not true for top-level nodes. The result of this is that every definition needs to be given twice, with only minor changes: once for top-level nodes, and once for nested nodes. In order to avoid this problem, we adopt the convention that **all top-level nodes are nested in a node with label "*root*"**. In this way, we can act as if all nodes are nested in exactly one other node:

$$\forall v \in V: \exists w \in V \cup \{root\}: (v,w) \in nested$$

Of course, in a visual representation we will never show the *root* node explicitly. Also, in the qualified label notation, if $(v,root) \in nested$ and $label(v) = A$ then we simply refer to this label as *A*.

V 4.2 CONTRACT TYPES IN THE PRESENCE OF NESTING

We will now investigate the impact of nesting on the primitive contract types, and define a number of new primitive and composite contract types that only make sense when dealing with nested graphs.

V 4.2.1 Extension and Cancellation

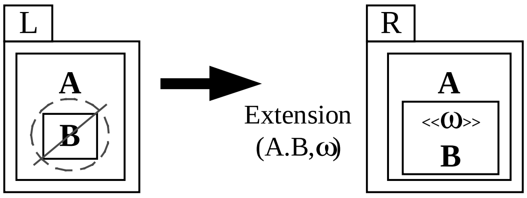
In the presence of nesting, the definitions of the primitive contract types *Extension* and *Cancellation* need to be modified slightly because they have to deal with the introduction and removal of nested nodes. Because of the orthogonal way in which the nesting mechanism was defined on top of labelled typed graphs (see Definition 18 of page 52), the required modifications will have little or no impact on the formal proofs in the previous chapter.

A nested *Extension* not only needs to mention the label and type of the newly introduced node, but additionally should be qualified by the label of the parent node in which it should be nested. Moreover, the *Extension* contract type should add a new pair to the *nested* relation. An analogous reasoning can be made for the *Cancellation*. Nesting does not have an impact on the other primitive contract types *Refinement*, *Coarsening*, *NodeRetyping* and *EdgeRetyping*.

Extension(A.B, ω)

Description. Add a new node *w* with label *B* inside an existing parent node *v* with label *A*, and add the pair *(w,v)* to the *nested* relation.

Definition:



Application conditions:

PreCond(Extension(A.B, ω)) = { v∈L, label(v)=A, ∄w∈L: label(w)=B and (w,v)∈nested }

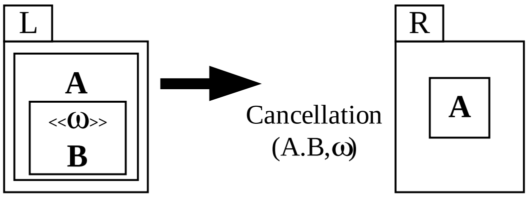
PostCond(Extension(A.B, ω)) = { v∈R, label(v)=A, w∈R, label(w)=B, type(w)=ω, (w,v)∈nested, Adjacent(w)=∅ }

Remark. Node *v* itself can already be nested in a node.

Cancellation(A.B, ω)

Description. Remove a nested node *w* with label *B* from a graph, as well as the *nested* relation between *w* and its parent node *v* with label *A*. This operation is only allowed if *w* contains no adjacent edges or nested nodes.

Definition:



Application conditions:

PreCond(Cancellation(A.B, ω)) = { v∈L, label(v)=A, w∈L, label(w)=B, type(w)=ω, (w,v)∈nested, Adjacent(w)=∅, ∄u∈L: (u,w)∈nested }

PostCond(Cancellation(A.B,ω)) = {v∈R, label(v)=A, ∄w∈R: label(w)=B and (w,v)∈nested}

From now on, **we will always use these new definitions of *Extension* and *Cancellation* instead of the former ones**.

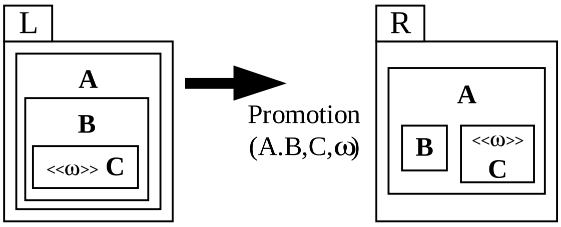
Note that all the primitive contract types we have defined until now are nesting preserving (Definition 19of page 53), because they do not change anything to existing *nested* relations. Even the new variants of *Extension* and *Cancellation* preserve the *nested* relation, although they can add new nested nodes, or remove existing nested nodes, as long as these nodes do not contain nested nodes themselves. In the following subsections we will see some new primitive contract types that are not nesting preserving.

V 4.2.2 Promotion and Demotion

A useful kind of primitive contract type that can be imagined in the presence of nesting allows us to move a particular node up (promote) or down (demote) the nesting hierarchy. *Promotion* of a nested node moves it to the same level as its parent, and takes all its ingoing and outgoing edges with it. *Demotion* is the inverse of *Promotion*, and moves a node inside a different node. All the edges to the demoted node remain fixed, however.

Promotion(A.B, C, ω)

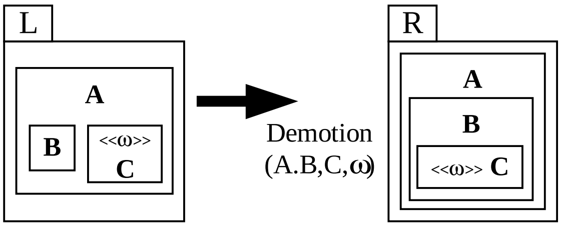
Description. Move a node with label *C* to the same level as the node with label *B* in which it is nested. Obviously, *B≠C* because of the injectivity constraints on the labels of nested nodes.
Definition:



Remark. All the edges that are adjacent to *C* will be preserved by the *Promotion*.

Demotion(A.B, C, ω)

Description. Move a node with label *C* one level *down* in the nesting hierarchy, by nesting it inside the node with label *B* which resides at the same level. Again, *B≠C* because of the injectivity constraints on the labels of nested nodes.
Definition:



Remark. All the edges that are adjacent to *C* will be preserved by the *Demotion*.

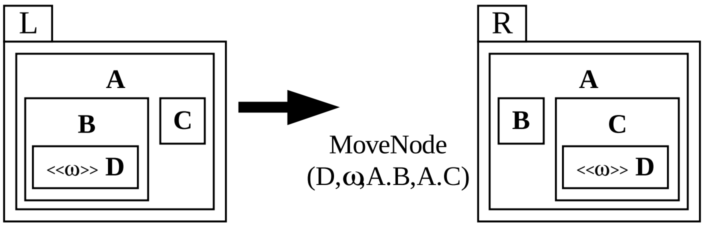
The above two primitive contract types are special in that they do not make any changes to the nodes or edges of a labelled typed graph. They only make a change to the nesting hierarchy between the nodes of the graph. The same is true for the primitive contract type *MoveNode* defined in the next subsection.

V 4.2.3 MoveNode

A final primitive contract type which does not preserve the nesting hierarchy consists of moving a nested node so that it becomes nested in a different node at the same level. A typical example of this is: moving attributes or operations in a class to a new class. These contract types are needed in some refactorings, as illustrated in [Tokuda&Batory98b].

MoveNode(D, ω, A.B, A.C)

Description. Move a node with label *D* so that it becomes nested in a different parent which resides at the same level as the original parent.
Definition:



Remark. All the edges that are adjacent to *D* will be preserved by *MoveNode*.

V 4.2.4 DeleteContents

A problem with the nested variant of *Cancellation* as introduced before is that it often is too restrictive in practice. It can only be applied if the node that is being cancelled is completely empty, i.e., it does not contain any nested nodes. Therefore, we will introduce a new composite contract type that allows us to delete the entire contents of a node by removing all its nested nodes as well as all edges between these nested nodes. The only restriction is that there should be no edges from outside the node to nested nodes or vice versa. If there are, these edges should be removed first.

DeleteContents(A, ω)

Description. Delete all nodes that are nested inside a node *v* with label *A* and type ω, as well as all edges between these nodes.

Definition: Let $N = \{ w \in V \mid (w,v) \in nested^{+} \}$ be the set of all nodes that are nested in *v*. If $\forall w \in N$: $Adjacent(w) \subseteq N$ (i.e., only edges between nested nodes are allowed) then we can define *DeleteContents* as the sequential composition of a *composite coarsening* that removes all edges between nodes of *N*, and a *composite cancellation* that removes all the nodes of *N*.

Remark. To ensure that all nested nodes and edges are removed, the above definition requires to take the transitive closure of the *nested* relation.

V 4.3 NESTING EVOLUTION CONFLICTS

The three primitive contract types *Promotion*, *Demotion* and *MoveNode* will not give rise to many new domain-independent evolution conflicts, because they do not change anything to the structure of the labelled typed graph to which they are applied. They only make changes to the nesting hierarchy. Therefore, only evolution conflicts can occur with other modifications that also change this nesting hierarchy. However, we will not discuss these conflicts in more detail. When customising the formal framework to a specific domain, more domain-specific conflicts can arise because of the introduction of domain-specific type constraints that impose additional restrictions on the nesting hierarchy. Examples of this will be given in the next chapter.

We will now take a closer look at the impact of the new definition of *Extension* and *Cancellation* on the possible applicability and evolution conflicts. As it turns out, the nested variants of *Extension* and *Cancellation* do not give rise to any new applicability conflicts. Concerning evolution conflicts we are not so lucky. Several new evolution conflicts can be defined in the presence of nesting. To this aim, we need to introduce a new contract type that indicates if a modification has been made to the internal details of a particular node with label *A* and type υ. Such a modification can be made in many different ways: by retyping the node *A*, by adding a new node inside *A* (*Extension*, *Demotion* or *MoveNode*), by removing a nested node from *A* (*Cancellation*, *Promotion* or *MoveNode*), by adding a new edge between nested nodes in *A* (*Refinement*), by removing an existing edge between nested nodes in *A* (*Coarsening*), by changing the type of a nested node (*NodeRetyping*) and by changing the type of a nested edge (*EdgeRetyping*). If we want to ignore the details of these internal changes, we make use of the *Modification* contract type:

Modification(A)

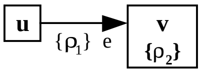
Description. Modifies the internal details of a node.

Definition: *Modification(A) = NodeRetyping(A,υ,ω)* or *Extension(A.B,ω)* or *Cancellation(A.B,ω)* or *Refinement(e,A.B,A.C,τ)* or *Coarsening(e,A.B,A.C,τ)* or *EdgeRetyping(e,A.B,A.C,τ,ϕ)* or *Promotion(C.A,B,ω)* or *Demotion(C.A,B,ω)* or *MoveNode(D,ω,C.A,C.B)* or *MoveNode(D,ω,C.B,C.A)* or *Modification(A.B)*

Remark. *Modification* is defined in a recursive way to deal with indirect nested nodes as well.

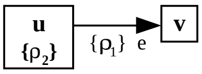
Using the *Modification* contract type, we can fine-tune the evolution conditions of section IV 4.4 in the presence of nesting. More specifically, we will define a nested variant of *EC6: Inconsistent target conflict* and *EC7: Inconsistent source conflict*:

EC_6': Inconsistent target conflict

Evolution condition. An inconsistent target conflict is detected by  with $\rho_1 \neq \rho_2$.

Occurrence. It occurs when a node with label *v* plays a role in both reuse contracts. Without nesting, this conflict could only occur when reuser ρ_1 performed a *Refinement(e,u,v,τ)*, *Coarsening(e,u,v,τ)* or *EdgeRetyping(e,u,v,ϕ,τ)* while reuser ρ_2 performed a *NodeRetyping(v,υ,ω)*. With nesting, however, the target node *v* can be modified in many more ways than a simple *NodeRetyping*, by making some changes to the internals of *v*. Each of these changes is described by *Modification(v)*.

EC_7': Inconsistent source conflict

Evolution condition. An inconsistent source conflict is detected by  with $\rho_1 \neq \rho_2$.

Occurrence. It occurs when a node with label *u* plays a role in both reuse contracts. Similar to the inconsistent target conflict, this conflict not only occurs when reuser ρ_1 performs a *Refinement(e,u,v,τ), Coarsening(e,u,v,τ)* or *EdgeRetyping(e,u,v,ϕ,τ)* while reuser ρ_2 performs a *NodeRetyping(u,υ,ω)*, but also when reuser ρ_2 performs an arbitrary *Modification(u)* to the internals of *u*.

The next conflict is not really a new conflict, but a higher-level view on existing conflicts that arise because of independent modifications that are made to nodes or edges that are nested inside the same parent node.

EC_8: Double node modification conflict

Occurrence. This conflict can be considered as a generalisation of the *AC7: Double node retyping conflict* in the presence of nesting. While a double node retyping occurs when two different persons try to modify the type of the same node, a similar conflict arises when two different persons modify the internals of the same node by means of a *Modification*. In some cases, usually when the same lower-level modification is performed twice, the double node modification conflict will coincide with existing applicability conflicts. In other cases it will coincide with a lower-level evolution conflict between nested nodes and edges.

V . 5 DOCUMENTING REUSE AND EVOLUTION

V 5.1 MOTIVATION

Until now, this dissertation focussed on using the graph formalism to *detect conflicts* between parallel evolutions of the same software artifact. However, one of the main benefits of the reuse contracts approach is that it can also be used to *document reuse and evolution*. In fact, this was the main focus in the Ph. D. dissertation of Carine Lucas [Lucas97]. If a software artifact is reused in different places of the software system, and all these "reuses" are documented by means of reuse contracts, it becomes possible to detect potential problems in these reused parts if the original software artifact is replaced by an evolved version. In this way, reuse contracts provide active support for dealing with the propagation of changes.

Documenting evolution is also essential if we want to integrate the reuse contract formalism in a *version management tool*. Many software configuration management systems use so-called *version graphs* to represent the version space [Conradi&Westfechtel98]. Each node in the graph represents a different version of the software, while edges represent the relationships between different versions. With *state-based* versioning, each version is defined as the state of an evolving item. With *change-based* versioning, a version is described in terms of changes applied to an item. This latter approach corresponds to reuse contracts, where evolution of a software artifact is expressed as a sequence of modifications to another artifact. Consequently, by explicitly representing each reuse contract as an edge between nested nodes, we obtain a kind of version graph.

In order to represent a reuse contract as an edge in a graph, the information contained in the modifier clause needs to be attached to the edge by putting it in the edge constraint. Because the edges must also be able to express composite contract types, we need to make some small extensions to the underlying graph formalism. This will be done in subsection V 5.2. The subsection that follows describes how reuse contracts can be represented explicitly as edges in a graph.

V 5.2 DERIVED EDGES

An important way in which the labelled graph formalism can be scaled up is by making use of so-called *derived edges*. Actually, a derived edge is an edge that can be expressed in function of other edges.

Formally, this can be dealt with by introducing a new relation *derived* $\subset E\times E$ on the edges of a labelled graph, in an analogous way as we have defined the relation *nested* $\subset V\times V$ on the nodes of a graph. If $(e,f) \in$ *derived*, we say that edge *e* is derived from edge *f*. Alternatively, we can use the notation *derived(e) = f* to express the same thing. The notation *derived(e)* $= \{e_1,e_2,\ldots,e_n\}$ is used if we have a one to many relation, i.e., if an edge *e* is is related to (read: derived from) many other edges.

Below, we will cover three special kinds of derived edges: *promoted* edges, *transitive closure* edges and *composite* edges.

V 5.2.1 Promoted Edges

The first kind of derived edge is only useful in the presence of a nested graphs. Whenever an edge exists between nodes, we can automatically derive an edge at a higher nesting level by replacing the source and target node by the nodes in which they are nested. We then say that the edge is "promoted" to a higher level. *Promoted edges* provide scalability, since they allow us to look at edges at a higher level of abstraction. Because the constraints on a promoted edge are not necessarily the same as on the edge which it is derived from, the promoted edge is given a new type *«promoted»* (or any subtype thereof).

Let *G* be a labelled graph with $v_i \in V_G \ \forall i\in\{1,\ldots,4\}$
$(e,v_1,v_2,$«promoted»$)$ is a **promoted edge** of *G* if $\exists (f,v_3,v_4) \in E_G$ such that
$(v_3,v_1) \in$ *nested*, $(v_4,v_2) \in$ *nested* and $(e,f) \in$ *derived*

Definition 55: Promoted edge

An example is given in Figure 31, where a *«promoted»*-edge *e* from *B* to *A* is derived from a *«uses»*-edge *f* between the nested nodes *B.D* and *A.C*. To show clearly that the *«promoted»*-edge *e* is derived from *f*, we have attached an additional constraint *{(f,D,C)}* to it.

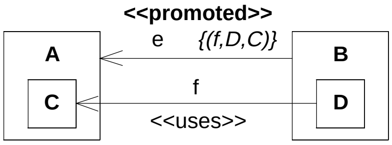


Figure 31: Promoted edge

Note that the above definition only allows promoted edges if both the source node and the target node are pulled up to a higher level. We could also consider extending this definition to allow promoted edges if only the source (or the target) of an edge is pulled up.

As the opposite of promoted edges, we could also define *demoted edges*, which are derived from an edge between nodes defined at a higher nesting level. For example, if the edge *e* from *B* to *A* in Figure 31 would be an ordinary edge, the edge *f* between subnodes *B.D* and *A.C* could be defined as a demoted edge which is derived from *e*.

V 5.2.2 Composite Edges

A promoted edge always corresponds to exactly one edge between lower-level nodes. If we want an edge to be derived from more than one other edge, we can use the notion of *composite edges*. Basically, a composite edge is an edge which is composed from (or derived from) a set of other edges, that can again be derived edges, if necessary.

The edge type *«composite»* is used to indicate that an edge is a composite. The relation *derived* is used to specify the edges a composite edge is built up from. An implicit constraint of a composite edge is that all the edges from which it is derived must have the same source and target node as the composite edge.

Let *G* be a labelled graph.
(e,v,w,«composite») is a **composite edge** of *G* if
$derived(e) = \{e_1,\ldots,e_n\}$ such that $n \geq 2$ and $\forall i \in \{1,\ldots,n\}: (e_i,v,w) \in E_G$.

Definition 56: Composite edge

V 5.2.3 Transitive Closure Edges

A final kind of derived edges are so-called *transitive closure edges*. In many situations, we are not only interested in *direct* dependencies of a particular type, but also in all *indirect* dependencies that can be found by following a sequence of edges of the same type. For this we can make use of the definition of transitive closure of a labelled graph, as given in Definition 10 on page 46. In order to distinguish transitive edges from ordinary ones, they will be given the edge type *«transitive»*. If desired, a constraint can be attached to the transitive edge, to specify the sequence of edges that gave rise to the transitive edge.

Let *G* be a labelled graph.
(e,v,w,«transitive») is a **transitive closure edge** of *G* if *(e,v,w)* is an edge of G^+.

Definition 57: Transitive closure edge

V 5.3 REUSE CONTRACTS AS EDGES

V 5.3.1 Evolution Edges

From a category-theoretical point of view, the idea of modelling evolution as an edge in a graph is very natural, since evolution is modelled by graph rewriting, and graph rewriting steps are formally defined as morphisms in a category with graphs as its elements. Moreover, a category is essentially nothing more than a sophisticated kind of graph, where the morphisms represent edges, and the elements represent nodes in the graph. For the interested reader, a meta-transformation of graph grammars has already been developed in [Parisi-Presicce96].

As a first step to express evolution between software artifacts as an edge between nodes in a graph, we need to encapsulate each graph (read: software artifact) that is subject to evolution in a node that represents this graph. This node will have the type *«graph»*. A graph *G* that evolves into a graph *H* is then represented by an *«evolution»*-edge from the *«graph»*-node with label *G* to the *«graph»*-node with label *H*. The constraint attached to this edge corresponds to the contract type.

For example, in Figure 32 (which again uses *Circle*, *Triangle*, *Geo* and *Point*) graph *G* evolves into graph G_2 by means of a primitive reuse contract *Refinement(vertices,Triangle,Point,«has-a»)*. This is modelled by a *«graph»*-node with label *G* containing the first graph, a *«graph»*-node with label G_2 containing the evolved graph, and an *«evolution»*-edge with constraint *{refinement=(vertices,Triangle,Point,«has-a»)}* between these two nodes. For each different primitive contract type, there will be a corresponding constraint.

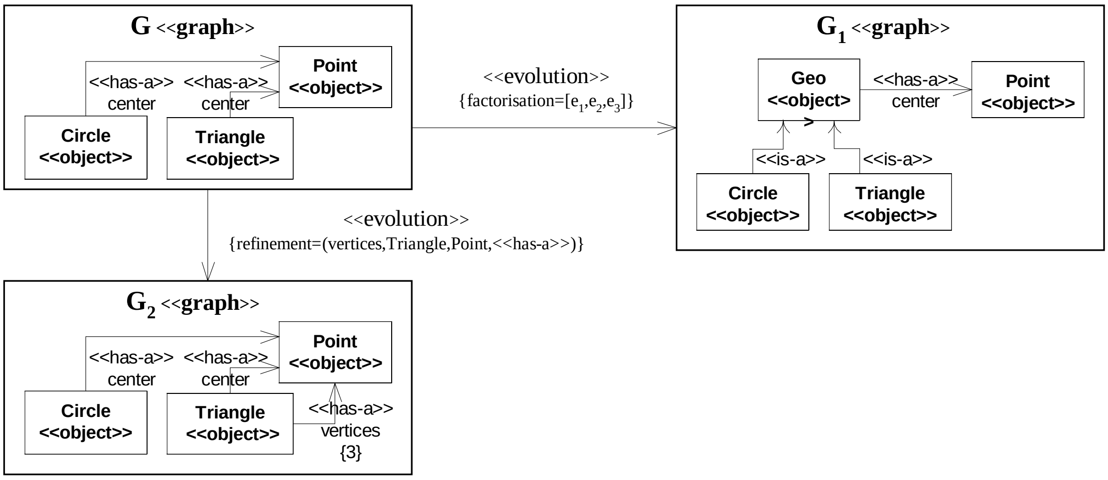
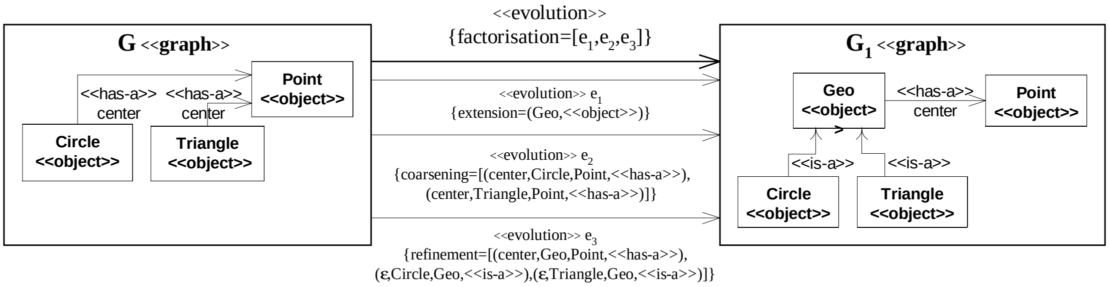


Figure 32: Graph representation of «evolution»-edges

Composite contract types can be modelled in exactly the same way, except that we have to use the notion of composite derived edges. This can be done by declaring *«evolution»* as a subtype of the *«composite»* edge type of section V 5.2.2. In Figure 32, an example of a composite *Factorisation* is modelled as a derived *«evolution»*-edge with constraint *{factorisation=[e_1,e_2,e_3]}* between *«graph»*-nodes *G* and G_1. The constraint list *[e_1,e_2,e_3]* attached to this composite edge specifies the set of primitive edges from which the composite edge is derived. The constraint additionally imposes an order on the primitive edges. Each of the edge labels e_i corresponds to a primitive evolution edge, representing a primitive contract type that is part of the composite contract type. This is shown in more detail in Figure 33, where the three primitive *«evolution»*-edges e_1, e_2 and e_3 corresponding to the composite *«evolution»*-edge are specified as well. These three primitive *«evolution»*-edges have constraints *{extension=…}*, *{refinement=…}* and *{coarsening=…}*, respectively.[4] Although additional constraints between these primitive edges can be attached as constraints to the composite *«evolution»*-edge, we have not done this as it would decrease the understandability of the example.



<<evolution>> {factorisation=[e1,e2,e3]}
G <<graph>>
<<has-a>> center
<<has-a>> center
Point <<object>>
Circle <<object>>
Triangle <<object>>
<<evolution>> e1 {extension=(Geo,<<object>>)}
<<evolution>> e2 {coarsening=[(center,Circle,Point,<<has-a>>), (center,Triangle,Point,<<has-a>>)]}
<<evolution>> e3 {refinement=[(center,Geo,Point,<<has-a>>), (ε,Circle,Geo,<<is-a>>),(ε,Triangle,Geo,<<is-a>>)]}
G1 <<graph>>
Geo <<object>>
<<has-a>> center
Point <<object>>
<<is-a>>
<<is-a>>
Circle <<object>>
Triangle <<object>>

Figure 33: Composite «evolution»-edge

[4] Actually, the edges e_2 and e_3 do not correspond to a primitive *Refinement* or *Coarsening*, but rather to a monotonous composite refinement and coarsening.

V 5.3.2 Reuse Edges

Explicitly documenting modification by means of reuse contracts is not only necessary when a software artifact *evolves* into a new version, but also when a software artifact is *reused* in another place. The way to deal with this is obvious: introduce a *«reuse»* edge type which has precisely the same characteristics as the *«evolution»* edge type.

A typical and wide-spread example of such *«reuse»*-edges can be found in any object-oriented class hierarchy. The inheritance mechanism is actually nothing more than an incremental modification mechanism that describes how each subclass reuses its parent class by making some modifications to it. The main disadvantage of inheritance is that the precise modifications between a subclass and its parent class are not explicitly documented. A subclass can add new operations or attributes, override existing operations with a new implementation, make some abstract operations concrete, or any combination of these. In [Steyaert&al96] it is explained how reuse contracts make inheritance more disciplined by explicitly documenting the inheritance link with reuse contract information. In our graph-based formalism, the same approach is taken, by modelling inheritance links as *«reuse»*-edges. The constraints on these edges document the exact modification that takes place.

When looking at version management systems [Conradi&Westfechtel98], the difference between *«evolution»*-edges and *«reuse»*-edges corresponds to the difference between *revisions* and *variants* of a version, respectively. A version that is intended to supersede its predecessor is called a *revision*. Revisions *evolve* along the time dimension for various reasons such as bug fixing and required changes in the functionality. Versions that are intended to coexist with the version from which they are derived are called *variants*. For example, a software artifact may support multiple operating systems, each representing a different variant.

V 5.3.3 Conflict Detection Revisited

Explicitly documenting all reuses and evolutions as edges in a graph makes it possible to automate the conflict detection algorithm. The approach goes as follows. Each time a software developer decides to make a modification to a particular part of the software system, represented as a graph nested in a node, a new *«evolution»*-edge that has this node as source is added to the graph. The reuse contract describing the actual modification is attached as a constraint to this edge. The *«evolution»*-edge is then compared against all other already existing *«evolution»*-edges and *«reuse»*-edges that have the same source node, to find out if and how the new evolution has an impact on existing reuses and evolutions. For each of these possibilities, the conflict detection algorithm is invoked to find the possible conflicts. If there are too many conflicts, the software developer might decide to abandon his intended evolution. Alternatively, a conflict resolution algorithm might be invoked to try and solve the detected conflicts.

Note that evolving part of a graph does not only have an impact on other parts that *directly* reuse (or evolve) this part. It can also affect all *indirect* reuses and evolutions. For example, in the case of an object-oriented class hierarchy, exchanging a parent class by a new version (*base class exchange*) can potentially have an impact on the entire inheritance hierarchy underneath this parent class, because all direct and indirect subclasses reuse this parent in some way. To deal with this, the *transitive closure* of all *«reuse»* and *«evolution»*-edges should be taken to find the impact of a particular evolution. In [Bohner&Arnold96b], a distinction is made between first-order impacts, second-order impacts, and in general N-th order impacts depending on the length of the path between the evolved element and its reuser.

V 5.3.4 Dealing with Nesting

The previous problems, whether they occur with reuse or evolution, become aggravated in the presence of nesting. Nesting makes it possible to evolve arbitrary subgraphs of a given graph. As a result, evolution conflicts will not only occur when this subgraph is evolved or reused by a second party, but can also arise at a higher or a lower level.

In order to see this more clearly, consider Figure 34. We start from a graph G_0 that contains a subgraph H_0 which on its turn contains a subgraph K_0. K_0 is reused somewhere else after making some slight modifications, leading to a new graph K_2. This is reflected by an edge *(r_2,K_0,K_2,«reuse»)*. Also, G_0 is evolved into a new version G_3, represented by an edge *(e_3,G_0,G_3,«evolution»)*.

Suppose that a software developer decides to evolve the subgraph H_0 into a new version H_1, without knowing that G_0 and K_0 have already been evolved and reused by K_2 and G_3, respectively. After this evolution, represented by an edge *(e_1,H_0,H_1,«evolution»)*, we need to investigate the potential impact on the existing reuse and evolution edges r_2 and e_3. Because the conflict detection algorithm can only compare different modifications of *the same* subgraph, we first need to perform the following steps.

- To see the impact of e_1 on r_2, which is defined at a *lower* level, we have to know how the evolution edge e_1 affects K_0. Therefore, a derived edge *(f_1,K_0,K_1,«evolution»)* will be calculated, which is nothing more than the restriction of e_1 to K_0. After that, the conflict detection algorithm can be applied to f_1 and r_2, since they represent two different modifications of the same initial graph.
- To see the impact of e_1 on e_3, which is defined at a *higher* level, we need to take the opposite approach. More specifically, the evolution edge e_3 needs to be restricted to H_0, leading to a derived edge *(f_3,H_0,H_3,«evolution»)*. This derived evolution edge f_3 then needs to be compared with e_1 to find any potential conflicts.

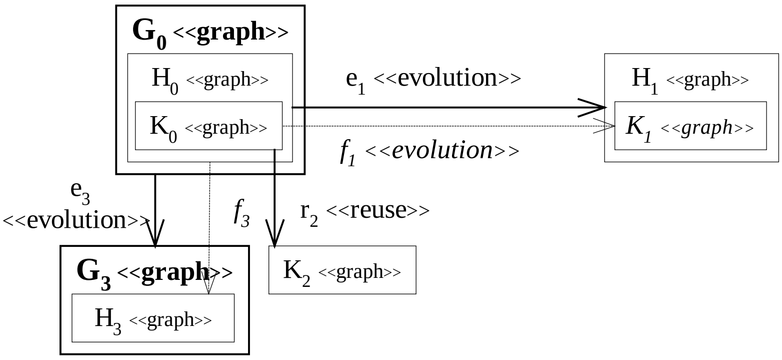


Figure 34: Detecting evolution conflicts in the presence of nesting

V . 6 OTHER EXTENSIONS

This section briefly discusses some other extensions that are useful to enhance the scalability of the formalism. Some of them should be made to the layer of the reuse contract formalism, e.g., the introduction of new kinds of primitive contract types such as relabelling. Others require changes to the underlying formalism of labelled typed graphs, e.g., by introducing parameterised nodes and edges.

V 6.1 MORE PRIMITIVE CONTRACT TYPES

While the primitive contract types *Extension*, *Cancellation, Refinement*, *Coarsening*, *NodeRetyping* and *EdgeRetyping* are sufficient do describe any possible graph modification, in practice they often give rise to complex sequences, even to describe relatively simple modifications. Although we can deal with this problem by using predefined composite contract types, this is not always desirable. For example, the only way to change the label of a node in our current formalism is by removing all adjacent edges from this node, removing the node, introducing a new node with the same type but a different label, and reintroducing all the edges on this new node. This is a lot of overkill for such a simple operation. Therefore, we propose to introduce a new primitive contract type that allows us to relabel a node or edge in a graph directly.

Another pair of useful primitive contract types allow us to replace the type of a node or edge by one of its subtypes. These contract types can be considered as type-safe variants of *NodeRetyping* and *EdgeRetyping*.

V 6.1.1 Relabelling

Until now, most of the primitive contract types that have been defined were label-preserving. In practice, however, one sometimes wants to change the label of a node or edge, without influencing the rest of the graph. For this, we need to introduce two new primitive contract types *RelabelNode* and *RelabelEdge*.

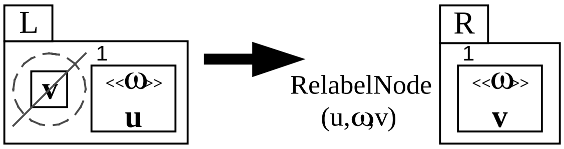
The reason why we haven’t dealt with relabellings before is a very pragmatical one: they were not mentioned in the existing work on reuse contracts [Steyaert&al96, Lucas97], nor where they needed to deal with any of the composite contract types introduced in [Lucas97]. Another reason is that we wanted to restrict our initial set of primitive contract types in order not to make the conflict detection process and normalisation algorithm too difficult to understand.

Nevertheless, we will take a closer look at relabelling now, and briefly discuss its impact on the evolution conflicts and the normalisation algorithm.

RelabelNode(u,ω,v)

Description. Change the label of a node from *u* to *v*.

Definition:



Application conditions:

PreCond(RelabelNode(u,ω,v)) = { (u,ω) ∈ L, v ∉ L }

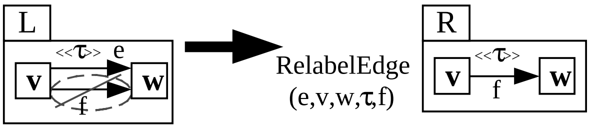
PostCond(RelabelNode(u,ω,v)) = { (v,ω) ∈ R, u ∉ R }

Remark. The above definition ensures that, if a node is relabelled, all edges arriving in, or leaving from this node, are preserved after the relabelling.

RelabelEdge(e,v,w,τ,f)

Description. Change the label of an edge from *e* to *f*.

Definition:



Application conditions:

PreCond(RelabelEdge(e,v,w,τ,f)) = { (e,v,w,τ) ∈ L, (f,v,w) ∉ L }

PostCond(RelabelNode(e,v,w,τ,f)) = { (f,v,w,τ) ∈ R, (e,v,w) ∉ R }

Although these two new primitive contract types are not *label preserving*, they are still *injective*. This is important, since the injectivity property (Property 12 of page 80) was required in many properties related to reuse contracts.

Like *Cancellation*, relabelling is a very intrusive operation. It causes *applicability conflicts* with all existing contract types. Below, the possible pairs (P_1, P_2) of primitive contract types that give rise to an applicability conflict are enumerated. A distinction is made between *RelabelNode* and *RelabelEdge*.

Without going in detail, P_1 = *RelabelNode(v,υ,w)* causes an applicability conflict in combination with each of the following primitive contract types P_2:

Extension(w,ω), Cancellation(v,υ),
Refinement(e,u,v,τ), Refinement(e,v,u,τ),
Coarsening(e,u,v,τ), Coarsening(e,v,u,τ),
RetypeNode(v,υ,ω), RetypeEdge(e,u,v,τ,φ), RetypeEdge(e,v,u,τ,φ),
RelabelNode(u,ω,w), $RelabelNode(v,\upsilon,v_2)$,
RelabelEdge(e,u,v,τ,f), RelabelEdge(e,v,u,τ,f)

In a analogous way, P_1 = *RelabelEdge(e,v,w,τ,f)* causes an applicability conflict in combination with each of the following primitive contract types P_2:

Cancellation(v,υ), Cancellation(w,ω),
Refinement(f,v,w,φ), Coarsening(e,v,w,τ),
RetypeEdge(e,v,w,τ,φ),
$RelabelNode(v,\upsilon,v_2)$, $RelabelNode(w,\omega,w_2)$,

$RelabelEdge(e_2,v,w,\phi,f)$, $RelabelEdge(e,v,w,\tau,e_2)$

Fortunately, the above conflicts can be resolved in an automatic way, so the user does not necessarily need to be aware of them. Whenever an applicability conflict occurs between P_1 = *RelabelNode(v,υ,w)* and one of the possibilities for P_2 mentioned above, it suffices to adapt P_2 so that it takes this relabelling into account. For example, if P_2 = *Refinement(e,u,v,τ)* we can replace it by a new primitive contract type P_2' = *Refinement(e,u,w,τ)*. An analogous reasoning can be made in all other cases.

There will be no new *evolution conflicts* that are introduced by relabellings. All relabelling conflicts can be detected as applicability conflicts. Because of this, the graph pattern approach for detecting evolution conflicts (as introduced in section IV 4.4.3) does not need to be revised in the presence of relabelling.

The *normalisation algorithm* and *extraction algorithm* also need to be revised if we want to take relabellings into account. Indeed, relabellings can be absorbed with other primitive contract types in various ways. Below, we give a list of absorbing pairs of primitive contract types to which relabellings can give rise. As in Definition 52 of page 119, (P_i, P_{i+1}) denotes the absorbing pair, while P_i' denotes the absorption contract type:

(P_i, P_{i+1}) = *(Extension(v,υ), RelabelNode(v,υ,w))* and P_i' = *Extension(w,υ)*
(P_i, P_{i+1}) = *(RelabelNode(v,υ,w), Cancellation(w,υ))* and P_i' = *Cancellation(v,υ)*
(P_i, P_{i+1}) = *(RelabelNode(u,υ,v), NodeRetyping(v,υ,w))* with *u≠w* and P_i' = *RelabelNode(v,υ,w)*
(P_i, P_{i+1}) = *(Refinement(e,v,w,τ), RelabelEdge(e,v,w,τ,f))* and P_i' = *Refinement(f,v,w,τ)*
(P_i, P_{i+1}) = *(RelabelEdge(e,v,w,τ,f), Coarsening(f,v,w,τ))* and P_i' = *Coarsening(e,v,w,τ)*
(P_i, P_{i+1}) = *(RelabelEdge(e,v,w,τ,f), RelabelEdge(f,v,w,τ,g))* with *e≠g* and P_i' = *RelabelEdge(e,v,w,τ,g)*

Using these absorbing pairs, the normalisation algorithm can be modified accordingly.

V 6.1.2 Subtyping

The primitive contract types *NodeRetyping* and *EdgeRetyping* were introduced in section IV 2.2 (page 78) to change the type of a node or edge. Usually, this is a dangerous operation, since changing the type of a node or edge might cause the type constraints, as specified in the type graph, to become invalidated. However, there is a special kind of retyping that is less problematic, namely when a type is replaced by one of its subtypes, as specified in the node type partial order and edge type partial order (see page 54). Indeed, since type constraints are inherited by subtypes (because of Definition 22 on page 54), the type constraints will not become invalidated.

SubtypeNode(v,υ,ω)

Description. Change the type of a node *v* from υ to a subtype ω as defined by the partial order *(NodeType, $\leq_V$)*.

Definition: *SubtypeNode(v,υ,ω) = NodeRetyping(v,υ,ω)* where $\omega \leq_V \upsilon$

SubtypeEdge(e,τ,ϕ)

Description. Change the type of an edge *e* from τ to a subtype ϕ as defined by the partial order *(EdgeType, $\leq_E$)*.

Definition: *SubtypeEdge(e,v,w,τ,ϕ) = EdgeRetyping(e,v,w,τ,ϕ)* where $\phi \leq_E \tau$

Both contract types specified above are subtype preserving, i.e., they are morphisms in the category ***LTGraphLT***$_s$, as defined on page 56, in section III 2.4.6.

V 6.2 Information Hiding

Although graphs are an intuitive formalism, often they have a tendency to become too complex. In those cases, we need mechanisms to *hide* unnecessary or unimportant details from the graph. These mechanisms, which are very useful for visualising graphs in tools, will not have an impact on the formal model. Elements that are hidden by a tool will still remain present in the underlying representation, and can be made visible again (if necessary) at any time.

V 6.2.1 Collapsing Nodes

One of the easiest ways to hide information from a graph is by *omitting the label or type of a node or edge* whenever these are not relevant.

A second way to remove unnecessary details is by *hiding all nodes and edges that are nested in a particular node*. We say that this node is *collapsed*. When a node is collapsed, there are two options for dealing with outgoing edges from, or incoming edges to, its nested nodes. One option is simply to omit these edges in the collapsed variant. In that case, however, it becomes impossible to see how the nested nodes depend on external nodes (or vice versa), since the dependencies have become invisible. Another alternative is still to show the dependencies at a higher level, by using the notion of *«promoted»*-edges as explained in section V 5.2.1. Many different edges at a lower level can correspond to the same promoted edge at a higher level.

V 6.2.2 Subsets of Nodes, Edges or Types

Another way to hide information in a graph is *by only considering a carefully selected subset of nodes and/or edges*. The selection of this subset could be made manually, using a visual tool, or could be made by relying on type information. For example, if we are only interested in nodes of a particular type (or a subtype thereof), we only need to display these nodes and all their interrelationships. Similarly, if we are only interested in particular types of edges, we could display all nodes, but with only edges of the specified type (or a subtype thereof) between them. A combination of both could be taken by simultaneously restricting the type of nodes and the type of edges that should be displayed.

For example, in an object-oriented class diagram we could decide to display the *classes* and *interfaces* only, as well as the *generalisation* relationships between them. In this way the inheritance relationships on classes and interfaces become clear.

V 6.3 GENERIC GRAPHS

In some cases, it would be useful to work with *generic graphs*, with pieces of the graph still left unspecified. For example, we could have a graph where the nodes have no label yet, or graphs where the type of some nodes still remains to be filled in. This is for example useful when we want our graphs to represent object-oriented frameworks, that can have abstract classes and abstract methods that do not yet have an implementation. Customising the framework corresponds to filling in the “holes”, usually called “hot-spots”. At a formal level, “customisation” of the corresponding generic graph would be achieved by filling in the partially defined nodes or edges.

Generic graphs could also be used to deal with all kinds of template mechanisms in object-oriented programming languages, such as template classes in C++. Even for UML, which is a standard object-oriented analysis and design notation, a template construct is defined as part of its underlying metamodel.

Some existing approaches towards graph rewriting allow one to deal with parameterisation or genericity to some extent. For example, PROGRES [Schürr95] allows for parameterised nodes, but does not yet support parameterised edges because of the unsolved type checking problems they give rise to. Because of this increased complexity, we have not dealt with generic graphs in this dissertation.

V . 7 Summary and Future Work

V 7.1 Summary

In this chapter we dealt with some scalability issues of the reuse contract formalism presented in the previous chapter. Scalability has been addressed in various ways.

By defining *composite contract types*, we showed how to deal with sequences of primitive contract types. We illustrated a number of useful *predefined* composite contract types. In some situations, the use of composite contract types allowed us to ignore particular evolution conflicts that occurred in its primitive constituents. This was for example the case with *Factorisation*, which was a behaviour-preserving transformation.

Another way in which we addressed scalability was by introducing a *normalisation algorithm*, which allowed us to remove redundancy in an arbitrary sequence of primitive contract types. This gave rise to a shorter but equivalent evolution sequence, thus making the conflict detection process more efficient.

We also investigated how to deal with evolution in the presence of *nested graphs*. This required us to make changes to existing primitive contract types, to define new primitive and composite contract types, and to specify new evolution conflicts.

By defining *«evolution»*-edges and *«reuse»*-edges, we managed to express graph modifications as edges in a graph. This enabled us to document reuse and evolution explicitly by means of reuse contracts, which allows us to provide better support for change propagation and impact analysis [Steyaert&al96, Lucas97].

Some other extensions that are necessary when applying the reuse contract formalism in practice were briefly discussed. First, we looked at new primitive contract types for dealing with relabelling and subtyping on nodes and edges. The relabelling contract types showed to be very dangerous, because they gave rise to many applicability conflicts. Secondly, we discussed some information hiding mechanisms, which are necessary when visualising graphs in tools. Instead of showing the entire graph visually, we explained some mechanisms to display only the essential or desired information. A final way to scale up the approach would be to make the underlying graphs more generic.

V 7.2 Future Work

Despite all these necessary extensions, there are still many technical issues that need to be resolved. Because of time limitations, we have not been able to investigate them in this dissertation, so these issues remain future work.

V 7.2.1 Refining the Normalisation Algorithm

In order to remove redundancy in an arbitrary sequence of primitive contract types, this chapter introduced a *normalisation algorithm*. This algorithm allowed us to remove redundant primitive contract types that did not contribute to the result obtained when applying the sequence as a whole. Although this is an important result, it still needs to be generalised in many ways.

- First of all, we need to know the impact of type constraints on the normalisation algorithm. In this chapter, we only looked at normalisation without taking type constraints into account. One can wonder if the normalisation algorithm remains valid in the presence of type constraints. Although we believe this to be the case, with only slight changes to the algorithm, it still needs to be checked.
- Secondly, we need to find out if the normalisation algorithm remains applicable when considering *composite contract types*. Normalisation has only been shown for primitive contract types, and is not necessarily scalable to the coarser-grained composite contract types. One solution to this problem is to “flatten” all composite contract types into sequences of primitive ones, but this leads to a loss of information. What’s even worse, after normalisation the “flattened” version of the composite contract type is likely to become scattered throughout the normalised reuse contract, thus making it virtually impossible to reconstruct the composite contract type after normalisation. In order to find a solution to this problem, clearly more research is needed. However, to be able to perform this research, we first need to have a significant number of useful predefined composite

contract types. To find out what are the most likely candidates for interesting and frequently occurring predefined composite contract types, we need to perform larger-scale experiments.

- As another generalisation, we need to find out what is the impact of the new primitive contract types *Promotion*, *Demotion*, *MoveNode*, *RelabelEdge*, *RelabelNode*, *SubtypeEdge* and *SubtypeNode* on the normalisation algorithm.
- The current version of the normalisation algorithm is rather inefficient, since it is similar to a bubble-sort algorithm. Trying to improve the efficiency of the algorithm is another interesting topic for future work.

V 7.2.2 Merging two Composite Reuse Contracts

While the previous chapter explained how to detect evolution conflicts when merging two primitive reuse contracts, this chapter investigated the case where *one* of both reuse contracts is *composite* instead of primitive. By using the normalisation algorithm, we could first minimise the composite reuse contract by removing all redundant contract types, and then detect conflicts with the primitive one.

In practice however, it is usually the case that *both* reuse contracts are *composite*. Then the question arises how two such sequences can be merged. A straightforward approach could be taken, by normalising both sequences, and then detecting evolution conflicts by comparing the primitive contract types in the remaining sequences. However, more intelligent algorithms could be devised to reduce the effort and make the conflict detection process more efficient.

One avenue of research which is worthwhile investigating would be to use a kind of *merge sort*. Given two composite contract types $G_0 \Rightarrow_{C1} G_1$ and $G_0 \Rightarrow_{C2} G_2$ that need to be merged, we could first decide to normalise C_1 and C_2 to a minimal sequence M_1 and M_2, respectively. Next, we could perform a kind of mergesort of these minimal sequences by first merging the composite extensions of M_1 and M_2, next merging the composite refinements of M_1 and M_2, and so on until all monotonous composite contract types of M_1 and M_2 are merged. If there are no applicability conflicts, the merge should give a new normalised composite contract type $G_0 \Rightarrow_{M3} G_3$ which should be equivalent to the sequential composition $G_0 \Rightarrow_{C1;C2} G_3$. The interesting question is how we can use this process to find possible evolution conflicts between M_1 and M_2 in an efficient way, without needing to investigate the entire result graph G_3.

V 7.2.3 Conflict Resolution Techniques

The main focus in this dissertation lies on formal techniques for *conflict detection*. Once the conflicts are detected, however, we also need automated assistance for *conflict resolution*. Although we do not think the conflict resolution process can be fully automated, the reuse contract formalism can help in this process in many ways. In [Mezini97] this issue is discussed in more detail.

V 7.2.4 Implementing the Conflict Detection Algorithm

We have already implemented a prototype version of the conflict detection algorithm in PROLOG, and the normalisation algorithm in Mathematica. Obviously, both algorithms should be combined. Therefore we need to revise and update our existing PROLOG implementation.

The new implementation should incorporate the improvements mentioned in the previous subsections: the normalisation algorithm should deal with type constraints, nesting should be dealt with, the new primitive contract types should be taken into account, and it should be possible to merge two independent composite contract types. Moreover, all of these issues should be dealt with in an *efficient* way. Preferrably, the new implementation should also provide some support for conflict resolution.

V 7.2.5 Related Tools

In practice, the reuse contract approach will rarely be used on its own, but needs to be integrated with existing tools that already provide partial support for managing software development. Below, we discuss three kinds of tools in more detail: version management systems, reuse repositories and CASE tools. In the ideal situation, all these tools are integrated into a single environment.

A *version management system* is probably the closest to the reuse contract approach. Each time a modification is made to an existing software artifact, a new version is created which incorporates these modifications. The old version is still available for future reference. Software developers may decide to make independent evolutions of the same version, and merge them in a later stadium. During this merging, the reuse contract approach can assist in identifying potential conflicts, to avoid having a merge that behaves in an undesired way. To be able to do this, all modifications between any two subsequent versions need to be documented with reuse contracts.

Often a *reuse repository* is employed in order to deal with reuse properly. Basically, this repository is a carefully chosen collection of reusable software artifacts. When developing a software system, artifacts from this repository can be reused by incorporating them in the specific application that is being developed. Sometimes, this reuse can be performed “as is”, but usually some modifications need to be made to allow the reusable artifact to work properly in the new context. Reuse contracts can be employed to document these modifications. In this way, evolution conflicts can be detected when elements from the repository, that are being reused in the application, evolve.

Reuse contracts should also be integrated in *CASE-tools*, i.e., tools that provide support during the analysis and design phases of the software life-cycle. Although reuse and evolution are considered more important during the early development phases, many existing CASE tools do not provide adequate support for reuse during these phases, and even less support for evolution. Yet, the presence of adequate change management mechanisms is recognised as an important prerequisite for successful reuse [Goldberg&Rubin95]. Therefore, we propose to integrate reuse contracts in existing CASE tools to address their current shortcomings. Small experiments with existing CASE tools have illustrated that this integration can be performed in a fairly straightforward way.

VI . Domain Independence of the Formalism

This chapter validates the domain-independence of our formal foundation for evolution, by showing how it can be customised to different domains such as collaborating classes, class diagrams and software architectures.

VI . 1 INTRODUCTION

VI 1.1 OVERVIEW

The previous chapter formally defined reuse contracts in terms of labelled typed graphs and conditional graph rewriting, and showed how to deal with evolution conflicts. Scalability issues were also addressed in various ways.

When reconsidering the original thesis statement, the only thing that remains to be validated is the *domain independence* of the formal framework. Although we did not rely on any domain-specific characteristics when defining the formalism, we still have to show that it can be customised to different domains, without needing to change the underlying formalism. This will be done in this chapter.

We will first verify if the formal model is general enough, by illustrating that it is an extension of existing work on reuse contracts, more specifically evolution in *class inheritance hierarchies* [Steyaert&al96] and evolution of *collaborating classes* [Lucas97]. Since the latter is already a generalisation of the former, we only need to show how the approach in [Lucas97] can be expressed in our formalism.

As a second part of the validation, we will show that our formal model for reuse contracts can be applied to new domains, such as evolution of analysis and design artifacts. For practical reasons, we restrict ourselves to UML [OMG97a], the industry standard for object-oriented analysis and design modelling. First, we illustrate how the formalism can be used to add support for evolution of *UML class diagrams*, probably the best understood and most widely used subset notation of UML. Next, we explain how these ideas can be applied directly to deal with evolution of other kinds of UML diagrams as well.

Finally, we sketch how the formal framework can be customised to a completely different domain, namely *software architectures*.

VI 1.2 APPROACH

Each time the domain-independent reuse contract framework is customised to a specific domain, the following steps need to be carried out:

- Identify the different types of nodes and edges that are needed to deal with the domain-specific concepts and their interrelationships. To make things easier, put all the different (domain-specific) node types in a *node type partial order*, and all the edge types in an *edge type partial order*, as explained in section III 2.4.3.
- Specify the *type constraints* between the different kinds of node types and edge types. Some types of nodes can only be nested in other nodes, some types of edges are only allowed between nodes of particular types, etc. Most of these type constraints can be expressed visually by means of a *type graph*, as explained in section III 2.4.2. These constraints can be regarded as extra well-formedness rules that are specific to the domain.
- Some *domain-specific type constraints* cannot be expressed visually in the type graph. Therefore, they have to be specified separately, preferrably in a formal way. Several examples of this were given in section III 2.4.
- As a following step, *domain-specific primitive contract types* need to be defined in terms of the predefined primitive contract types *Extension*, *Cancellation*, *Refinement*, *Coarsening*, *NodeRetyping*, *EdgeRetyping*, etc. Some of the domain-independent primitive contract types might not be required, while others might only be needed for particular kinds of node or edge types. For each domain-specific primitive contract type, a new name must be chosen that makes more sense in the considered domain.
- Of course, not every domain-specific contract type will be immediately expressible in terms of a single primitive contract type. For more complex operations, it might be required to define them as *domain-specific composite contract types*, in terms of a number of more elementary contract types.

Usually, additional constraints will be needed to specify how the constituents of a domain-specific composite contract type work together.

- Once the domain-specific contract types have been identified, we can fine-tune the *applicability conflicts*. These conflicts will also be given a new name that is meaningful in the domain. Moreover, the domain-specific type constraints that were introduced can give rise to some extra domain-specific applicability conflicts.
- Concerning the *evolution conflicts*, we need to specify which of the domain-independent evolution conflicts are relevant in the domain, and which ones can be ignored entirely or partially. Some evolution conflicts may only occur for particular types of nodes or edges. Depending on the domain, some conflicts might be less important than others. Again, a clear name should be given to each of the selected evolution conflicts.
- If necessary, a closer look can be taken at the domain-specific composite contract types, to see if they absorb particular evolution conflicts (as was the case with the behaviour-preserving *Factorisation* in section V 3.1).
- Finally, *domain-specific resolution techniques* should be specified that allow us to resolve the domain-specific evolution conflicts in a semi-automated way.

VI . 2 COLLABORATION REUSE CONTRACTS

This section shows how our formalism of reuse contracts can be used to express evolution of class collaborations as introduced in [Lucas97]. This serves two purposes. First, it allows us to conclude that our formalism is more general than the one proposed in [Lucas97]. Second, it serves as a first customisation of the formal framework to a specific domain.

VI 2.1 INFORMAL DISCUSSION

In first generation class-based object-oriented languages, classes were the only units of reuse. Quickly a need arose for higher-level program entities that capture the patterns of *collaborations* between several classes. The absence of such higher-level class collaborations as first-class entities in an object-oriented programming language makes large object-oriented programs difficult to understand and maintain, because functionality is spread over several methods and it becomes difficult to understand "the big picture". Indeed, there is a common conviction in the object-oriented research community that class or object collaborations are essential to describe object behaviour. One of the main virtues of using object collaborations is nicely stated in [Reenskaug&al96]:

> *An isolated object cannot do anything because a message must have both a sender and a receiver. It is only when we consider structures of collaborating objects that we can study cause and effect, and reason about the suitability and correctness of objects and their structures.*

Another important advantage of using object collaborations is that it facilitates *separation of concerns*, since the different roles played by objects can be described by means of different collaborations. In this way, the complexity of a software system can be reduced by looking at the important aspects only.

There are numerous ways of dealing with object collaborations. Below, a number of alternative approaches that can be found in the research literature are mentioned. Some of these approaches focus on programming constructs in the implementation phase, while others are more devoted to the analysis and design phases. Please note that the enumeration of different approaches is not at all intended to be complete, but is merely mentioned to stress the importance of collaborations.

- At implementation level, *interaction contracts* are introduced in [Helm&al90] as a language construct to describe the interaction between a set of collaborating classes. Such a contract between the different participants in the collaboration describes how the different participants interact by means of behavioural dependencies. This description consists of a number of class interfaces for the different participants, as well as preconditions and invariants that need to be satisfied by the participants. An important feature of this language construct is its high level of formality. [Mezini&Lieberherr98] build further on the idea of interaction contracts by introducing *adaptive plug-and-play components* as an explicit language construct for expressing collaborative behaviour that involves a set of participants (classes) in an object-oriented application domain. It extends the standard object-oriented model in an orthogonal way.
- In [Reenskaug&al96], *role models* are introduced as descriptions of a structure of co-operating objects along with their static and dynamic properties. A role model describes the subject of the object interaction, the relationships between objects, the messages that each object may send to its collaborators, and the model information processes. While role models are essentially based on the same ideas as interaction contracts, they appear during the analysis and design phase, and have a graphical representation.
- Collaborations and interactions in UML [OMG97b] are the semantic equivalent of role models. *Collaborations* are used to describe the context of an object collaboration. To each collaboration, a set of *interactions* correspond that describe the message sending behaviour of the collaborating objects. The same collaboration can be used as context of many different interactions. These interactions are specified by means of *collaboration diagrams* or *sequence diagrams*, two alternative views on the same information.

VI 2.2 Class Collaborations

Following the general tendency in the object-oriented research community, so-called *collaboration reuse contracts* were used in [Lucas97] and [DeHondt98] for dealing with collaborating classes. When compared to other approaches, the main focus lies on adding support for evolution to class collaborations during the design phase.

[Lucas97] is a direct extension of [Steyaert&al96], where reuse contracts were introduced to deal with the fragile base class problem in class inheritance hierarchies. Because of this, we will only show that our formalism is a generalisation (and formalisation) of the reuse contract model proposed in [Lucas97].

The model of *collaborations* presented in [Lucas97] is fairly primitive. Basically, a collaboration consists of a set of *participants*, each consisting of a name, a set of *acquaintance relationships* (called the *acquaintance clause*) and a set of *operations* (called the *client interface*). The operations themselves contain a *specialisation clause*, representing the *invocations* performed by the operation. Note that these operation invocations are only allowed under particular well-formedness constraints. The operations to which they refer must actually exist, and an acquaintance relationship must be defined between the participants in which the operations that invoke each other reside. The set of all operations of a participant, together with the invocations they perform, is a kind of extended interface, which is called the *specialisation interface*. (The term specialisation interface originally comes from [Lamping93], where it was used in a more restricted way to indicate all self sends performed by an operation in a class.)

An example of a collaboration is shown in Figure 35. It illustrates the basic design of the Model-View-Controller as can be found in Smalltalk. There are three participants: *Model*, *View* and *Controller*. *View* and *Model* are mutually acquainted, and similarly are *View* and *Controller*. There is also a directed acquaintance relationship from *Controller* to *Model*. *View* contains only one operation, called *update*. *Model* also contains one operation, called *changed*, which invokes *update*. Finally, *Controller* contains three different operations *hasControl*, *activate* and *deactivate*, and *activate* invokes *hasControl*. Note that this operation invocation requires an acquaintance relationship to be present from *Controller* to itself. By convention, each participant is acquainted to itself, using the implicit acquaintance called *self*.

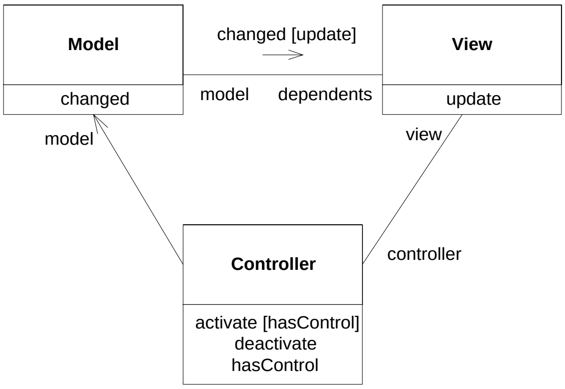


Figure 35: Model-View-Controller class collaboration

An extension of the basic notion of collaborations, which is also discussed in [Lucas97], concerns the introduction of *abstract and concrete operations*. The difference between both is that an abstract operation does not have a corresponding implementation, while a concrete operation does. In a similar vein, we can define a *concrete participant* as a participant where all its operations are concrete, while this is not the case for an *abstract participant*.

In the next section, we will show how this notion of collaborations can be expressed using the formalism of labelled typed graphs.

VI 2.3 Type Constraints

This section explains how domain-dependent information can be added to the formal framework by using the techniques explained in section III 2.4 of page 53. More specifically, we identify a domain-specific node type partial order and edge type partial order, as well as a type graph and additional type constraints to represent domain-specific well-formedness rules.

VI 2.3.1 Type Partial Orders

In order to represent collaborations as typed graphs, we need to distinguish the following kinds of node types. First, a distinction needs to be made between *«participant»*-nodes and *«operation»*-nodes. Second, both kinds of nodes can be further subdivided in an abstract and concrete variant. This leads to four new node types: *«conc-part»*, *«abs-part»*, *«conc-op»*, *«abs-op»*. Concerning the edge types, we only need to make a distinction between *«acquaintance»*-edges and *«invocation»*-edges. The partial orders that specify the subtype relationship for all these node types and edge types is presented in Figure 36.

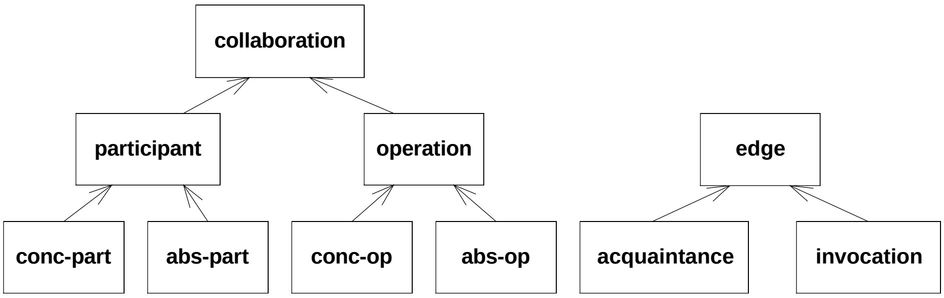


Figure 36: Collaboration node and edge type partial order

Using these types, the example of Figure 35 can be transformed into the graph representation of Figure 37. We take the convention of attaching an empty label ε to an *«invocation»*-edge from operation *m* to operation *n*.

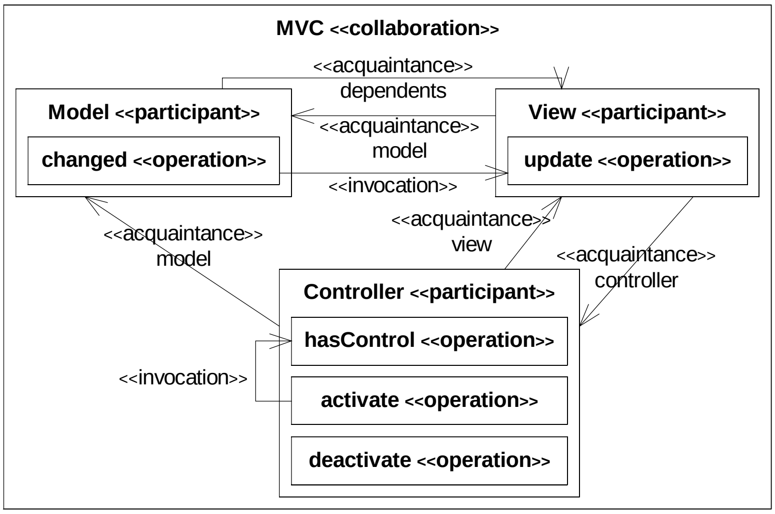


Figure 37: Nested graph representation of Model-View-Controller collaboration

VI 2.3.2 Type Graph

The exact relation between node types and edge types can be described by making use of the type graph specified in Figure 38. This type graph can be considered as a *metagraph* that enforces constraints on all its instances, i.e., on all graphs representing a class collaborations. For reasons of brevity, we have immediately added multiplicity constraints to this type graph, using an UML-like notation.

An *«operation»*-node should *always* be nested in exactly one *«participant»*-node, while any number of operations may be nested in the same participant. This is denoted by a *nested* edge with a `1` at the side of the *«participant»*-node and `*` at the side of the *«operation»*-node. A *«participant»*-node must always be nested in a *«collaboration»*-node. Keep in mind that, unlike the other edges in the type graph, the edges with label *nested* do not correspond to an edge type. Instead, they put an additional constraint on the nesting hierarchy which is formally represented as a relation $nested \subset V \times V$ between nodes of the graph. This was explained in section III 2.4.4 of page 54.

An *«invocation»*-edge is only allowed between *«operation»*-nodes, and an *«acquaintance»*-edge is only allowed between *«participant»*-nodes. An operation does not necessarily need to invoke an other operation, and it can invoke more than one other operation (indicated by a * at the target side of the *«invocation»*-edge). A participant can have any number of acquaintance relationships, but each of these relationships must connect them to exactly one participant.

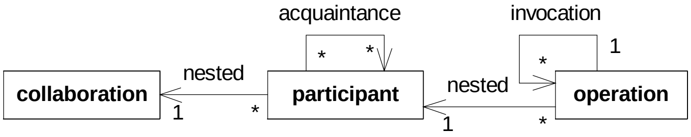


Figure 38: Collaboration type graph

No extra constraints are needed for the abstract or concrete versions of participants and operations, which is the reason why these are not mentioned in the type graph. All their constraints are "inherited" from the ones mentioned above. For example, an abstract operation is also allowed to invoke any number of other operations, whether they are abstract or not. This might seem awkward, since an abstract operation does not have a corresponding implementation, and hence cannot perform any invocations. However, invocations specified in an abstract operation should be interpreted as "desired" invocations that should be present once the operation becomes filled in with a concrete implementation. This allows us to express part of the desired behaviour of an abstract operation in a very simple way, without needing to resort to sophisticated formal specification techniques.

Note that the type graph expresses all edges that are allowed to occur. Absence of particular edges means that these are not allowed in concrete collaboration graphs. For example, a *«participant»*-node can never be nested in any other node, or an *«operation»*-node cannot be acquainted to any other node.

VI 2.3.3 Additional Type Constraints

There are two additional constraints that cannot be expressed immediately in the type graph of Figure 38. The first constraint (C_1) expresses that *operation invocations are only allowed if an acquaintance relationship is defined between the participants in which these operations reside*. The second constraint (C_2) expresses that *a concrete participant can only contain concrete operations*. The intuitive reason for this is that a concrete participant corresponds to an instantiable class with all the implementations of its methods filled in.

Constraint C_1: *«invocation»*-edge:

$\forall(\varepsilon,p.m,q.n,\text{«invocation»})\in E$: $p=q$ or $\exists(a,p,q,\text{«acquaintance»})\in E$

Constraint C_2: «conc-part»-node:

$\forall(p,\text{«conc-part»})\in V$: if $v\in V$ with $(v,p)\in nested$ then $type(v)=\text{«conc-op»}$

The reason for $p=q$ in constraint C_1 is that operation invocations within the same participant do not require an explicit *«acquaintance»*-edge, since every participant is assumed to know itself. Alternatively, this may be solved by explicitly defining an *«acquaintance»*-edge called *self* on each participant.

A constraint similar to C_2 is not necessary for abstract participants. An abstract participant is allowed to contain concrete operations only, although it will usually contain one or more abstract operations.

All type constraints mentioned above should be regarded as extra *well-formedness rules* on a graph. In other words, we have to restrict ourselves to the subset of all graphs that satisfy these well-formedness rules.

With respect to the detection of applicability and evolution conflicts, the effect of these additional well-formedness rules is fairly straightforward. They serve as invariants that need to be satisfied before and after each application of a primitive contract type. As a result, the set of possible contract types that is applicable to a particular graph is reduced. Moreover, these well-formedness constraints will give rise to extra *domain-specific applicability conflicts* in those cases were the combination of two independent modifications leads to a breach of one of the constraints. We will see some concrete examples of this in section VI 2.5.

VI 2.4 Domain-Specific Contract Types

This section explains how the domain-specific modification operations for collaborating classes can be expressed in terms of our primitive or composite domain-independent contract types.

VI 2.4.1 Primitive Contract Types

All the modification operations (so-called *reuse operators*) defined in [Lucas97] can be expressed easily in terms of the primitive contract types *Extension*, *Cancellation*, *Refinement*, *Coarsening*, *NodeRetyping* and *EdgeRetyping*. Note that we will always use the nested variants of *Extension* and *Cancellation* presented in section V 4.2.

All modification operations will be discussed using the following template:

Name of the modification operation according to [Lucas97]

> **Explanation.** Brief description of what the operation is intended to do.
> **Corresponds to:** Description of the primitive (or composite) contract type to which this operation corresponds in our formalism.
> **[optional] Remark.** Any additional remarks about the modification operation can be mentioned here. Mostly these are comments about how the well-formedness constraints restrict the applicability of the considered operation.

Observe that defining the modification operations of [Lucas97] in terms of our primitive contract types simplifies the work that needs to be performed substantially (which was one of the main reasons for defining a domain-independent formalism of reuse contracts in this dissertation). Instead of needing to define all domain-specific operations from scratch, we only need to specify how they can be expressed in terms of our primitive domain-independent contract types.

ContextExtension(C.p)

> **Explanation.** Adding a participant *p* to a collaboration *C*.
> **Corresponds to:** *Extension(C.p, «participant»)*
> **Remark.** The fact that *C* is a *«collaboration»*-node follows from the *nested* edge in the type graph.

ContextCancellation(C.p)

> **Explanation.** Removing a participant *p* from a collaboration *C*.
> **Corresponds to:** *Cancellation(C.p, «participant»)*

In the remainder, we implicitly assume that all participants are nested in the same collaboration *C*. In other words, when we write *p* (where *p* is the label of a participant), we actually mean *C.p*.

ContextRefinement(a, p, q)

> **Explanation.** Adding an acquaintance relationship *a* from participant *p* to participant *q*.
> **Corresponds to:** *Refinement(a, p, q, «acquaintance»)*
> **Remarks.**
> It is allowed to have more than one acquaintance relationship between the same participants, as long as these acquaintance relationships have different names.
> The fact that *p* and *q* are participants is ensured by the constraint in the type graph that guarantees that *«acquaintance»*-edges are only allowed between *«participant»*-nodes.

ContextCoarsening(a, p, q)

> **Explanation.** Removing an acquaintance relationship *a* between participants *p* and *q*.
> **Corresponds to:** *Coarsening(a, p, q, «acquaintance»)*
> **Remark.** Removing an acquaintance relationship is only allowed if there are no operation invocations defined over this relationship. If there are, removing the acquaintance would lead to a breach of the well-formedness condition imposed by constraint C_1.

ParticipantExtension(p.m)

Explanation. Adding a new operation *m* to participant *p*.

Corresponds to: *Extension(p.m, «operation»)*

Remark. Because of the subtyping relationship, the operation *m* that is added can be abstract or concrete. The participant *p* to which the operation is added can also be abstract or concrete. The case where the participant is concrete, while the added operation is abstract, is not allowed since it would break well-formedness constraint C_2.

ParticipantCancellation(p.m)

Explanation. Removing an operation *m* from participant *p*.

Corresponds to: *Cancellation(p.m, «operation»)*

Remark. Both the participant *p* and operation *m* can be abstract or concrete. Each of the possibilities is allowed. There are no breaches of constraints.

ParticipantRefinement(p.m, q.n)

Explanation. Adding an invocation of operation *q.n* by operation *p.m*.

Corresponds to: *Refinement(ε, p.m, q.n, «invocation»)*

Remarks.

Because of constraint C_1, an invocation from *p.m* to *q.n* can only be added if there is already an acquaintance relationship present from *p* to *q*.

Unlike with acquaintance relationships, there can be at most one operation invocation between any two operations *m* and *n* because of the multiplicity constraints in the type graph. By convention, this *«invocation»*-edge always has an empty label *ε*.

ParticipantCoarsening(p.m, q.n)

Explanation. Removing an invocation of operation *q.n* by operation *p.m*.

Corresponds to: *Coarsening(ε,, p.m, q.n, «invocation»)*

ParticipantConcretisation(p.m)

Explanation. Making an abstract operation *p.m* concrete by filling in its implementation details.

Corresponds to: *NodeRetyping(p.m, «abs-op», «conc-op»)*

Remark. Note that, because we deal with operations at design level rather than implementation level, the actual implementation code of a concrete operation is not explicitly modelled in the graph. If desired, however, it could be added by specifying the implementation in the constraint set of the *«conc-op»*-node.

ParticipantAbstraction(p.m)

Explanation. Making a concrete operation *p.m* abstract by removing its implementation.

Corresponds to: *NodeRetyping(p.m, «conc-op», «abs-op»)*

ContextConcretisation(p)

Explanation. Making an abstract participant *p* concrete.

Corresponds to: *NodeRetyping(p, «abs-part», «conc-part»)*

Remark. Because of constraint C_2, a *ContextConcretisation* is only applicable to a participant *p* if all of its nested operations are already concrete. If this is not the case, the *NodeRetyping* would breach the well-formedness condition imposed by C_2.

ContextAbstraction(p)

Explanation. Making a concrete participant *p* abstract.

Corresponds to: *NodeRetyping(p, «conc-part», «abs-part»)*

Remark. This primitive contract type was not considered in [Lucas97], but we mention it nevertheless since it is the obvious counterpart of the previously defined *ContextConcretisation*.

VI 2.4.2 Composite Contract Types

Some of the composed operations of [Lucas97] (like *ExtendingRefinement* and *ConnectedExtension*) have already been described earlier in a domain-independent way, in terms of composite contract types. We will only give one illustration of a domain-specific composite contract type here.

CompleteParticipantConcretisation(p)

> **Explanation.** Making an abstract participant *p* concrete, after having made each of its operations concrete.
> **Corresponds to:** *[ParticipantConcretisation(p.m$_1$), …, ParticipantConcretisation(p.m$_n$), ContextConcretisation(p)]* where *{p.m$_1$,…,p.m$_n$}* are the only abstract operations nested in *p*.
> **Remark.** This domain-specific composite contract type is defined in terms of the domain-specific primitive contract types *ParticipantConcretisation* and *ContextConcretisation*.

VI 2.5 DOMAIN-SPECIFIC CONFLICTS

In this subsection we reconsider the different kinds of evolution conflicts that were treated in [Lucas97], and systematically show that they all correspond to one of the applicability conflicts of section IV 3.2, or one of the evolution conflicts of section IV 4.4. Because in [Lucas97] no explicit distinction was made between applicability and evolution conflicts, we will treat them all in the same way here. If desired, one can skip this section on first reading, and go immediately to the discussion in the next subsection where the results will be summarised.

All conflicts will be explained using the following template:

Name of the conflict according to [Lucas97]

> **Occurs when.** Description of the two domain-specific modification operations that lead to the conflict under consideration. We will *not* explain *why* this combination leads to a conflict, since this has already been discussed in detail in [Lucas97], and it would require too much space to repeat it here.
> **Corresponds to:** Here we mention the name and number of the corresponding conflict in our dissertation. *AC$_i$* refers to one of the *applicability conflicts* discussed in section IV 3.2, while *EC$_i$* refers to one of the *evolution conflicts* discussed in section IV 4.4.
> **[optional] Remark.** Any additional remarks about the conflict can be mentioned here.

The first two conflicts we consider arise when different evolvers add an element with the same name. While in [Lucas97] a distinction was made based on the type of element that was added (either participants or operations), in our formalism both variants are detected in exactly the same way.

Participant Name Conflict

> **Occurs when.** Two *ContextExtensions* introduce the same participant *p*.
> **Corresponds to:** *AC1: Duplicate node conflict,* because two nodes *(p, «participant»)* with the same label are introduced by different *Extensions*. Indeed, a *ContextExtension* is defined in terms of an *Extension*.

Operation Name Conflict

> **Occurs when.** Two *ParticipantExtensions* of the same participant *p* introduce the same operation *m*.
> **Corresponds to:** *AC1: Duplicate node conflict*, because two nodes *(p.m, «operation»)* with the same label are introduced by different *Extensions*. Indeed, a *ParticipantExtension* is defined in terms of an *Extension*.

A second kind of conflict arises when an element is removed by one modifier, while a different modifier relies on the presence of this element. In retrospect, [Lucas97] made an unnecessary distinction based on the type of element that is removed. On the other hand, each of the following conflicts can arise in different situations, depending on how the second modifier relies on the presence of the element removed by the first modifier. In our formal model, each of these situations is regarded as a different kind of conflict for reasons of orthogonality.

Dangling Participant Conflict

Occurs when:

1. A *ContextCancellation* removes a participant *p* while a *ContextRefinement* adds a new acquaintance relationship *a* with this participant *p* as source or target.
2. A *ContextCancellation* removes a participant *p* while a different *ContextCancellation* also removes this participant *p*.
3. A *ContextCancellation* removes a participant *p* while a *ContextAbstraction* or *ContextConcretisation* makes the participant abstract or concrete.
4. A *ContextCancellation* removes a participant *p*, while a *ContextCoarsening* removes an acquaintance relationship *a* with *p* as source or target.

Corresponds to:

1. The first case corresponds to *AC3: Undefined source conflict* or *AC4: Undefined target conflict*, since the node *(p, «participant»)* is removed by a *Cancellation*, while a *Refinement* introduces an *«acquaintance»*-edge *a* with this node *p* as source or target.
2. The second case corresponds to *AC2: Double cancellation conflict*.
3. The third case corresponds to *AC9: Undefined node retyping conflict*.
4. The fourth case corresponds to an evolution conflict rather than an applicability conflict. More specifically, we have an *EC6: Inconsistent target conflict* or *EC7: Inconsistent source conflict*, depending on whether the *«acquaintance»*-edge *a* that is removed has *p* as source or target.

Dangling Operation Conflict

Occurs when:

1. A *ParticipantCancellation* removes an operation *p.m* while a *ParticipantRefinement* adds a new invocation with this operation *p.m* as source or target.
2. A *ParticipantCancellation* removes an operation *p.m* while a different *ParticipantCancellation* also removes this operation.
3. A *ParticipantCancellation* removes an operation *p.m* while a *ParticipantAbstraction* or *ParticipantConcretisation* makes this operation abstract or concrete.
4. A *ParticipantCancellation* removes an operation *p.m* while a *ParticipantCoarsening* removes an invocation with *p.m* as source or target.

Corresponds to:

1. The first case corresponds to *AC3: Undefined source conflict* or *AC4: Undefined target conflict* because node *(p.m, «operation»)* is removed by a *Cancellation*, while a *Refinement* introduces an *«invocation»*-edge with this node *p.m* as source or target.
2. The second case corresponds to *AC2: Double cancellation conflict*.
3. The third case corresponds to *AC9: Undefined node retyping conflict*.
4. The fourth case corresponds to an evolution conflicts rather than an applicability conflict. More specifically, we have an *EC6: Inconsistent target conflict* or *EC7: Inconsistent source conflict*, depending on whether the *«invocation»*-edge that is removed has *p.m* as source or target.

The next kind of conflict arises when different modifications add relationships with the same source element. Again, [Lucas97] made a distinction based on the kind of relationship (either *«acquaintance»* or *«invocation»*). In our formalism this distinction becomes unnecessary. On the other hand, a distinction needs to be made depending on whether or not both newly introduced relationships have the same name. If this is the case, we have an applicability conflict, otherwise it is an evolution conflict.

Operation Invocation Conflict

Occurs when: A *ParticipantRefinement* adds a new invocation from an operation *p.m* to a different operation, while an other *ParticipantRefinement* also adds an invocation from the operation *p.m* to a different operation. The conflict also occurs when one or both modifications is a *ParticipantCoarsening* which removes an invocation from operation *p.m* to a different operation.

Corresponds to:

If both *ParticipantRefinements* add the *same* operation invocation, we get applicability conflict *AC5: Duplicate edge conflict*, because the same edge is introduced twice. In the case where both modifications are *ParticipantCoarsenings* of the same operation invocation, we get an *AC6: Double coarsening conflict.*

If both modifications add or remove a *different* operation invocation, i.e., two *«invocation»*-edges with a different target node (but the same source node), we get evolution conflict *EC3: Double source conflict*, independent of whether we used *ParticipantRefinement* or *ParticipantCoarsening*.

Acquaintance Relationship Conflict

Occurs when: A *ContextRefinement* adds a new acquaintance relationship *a* from participant *p* to participant *q*, while an other *ContextRefinement* also adds an acquaintance relationship from participant *p* to a different participant. The conflict also occurs when one or both modifications is a *ContextCoarsening* which removes an acquaintance relationship from participant *p* to a different participant.

Corresponds to:

If both *ContextRefinements* add an acquaintance relationship with the *same* name to the *same* participant *q*, we get applicability conflict *AC5: Duplicate edge conflict*. In the case where both modifications are *ContextCoarsenings* of an acquaintance with the same name and the same participant *q*, we get an *AC6: Double coarsening conflict*

If both modifications (*ContextRefinement* or *ContextCoarsening*) add or remove an acquaintance relationship with a *different* name to the *same* participant *q*, we get evolution conflict *EC2: Double reachability conflict*. If both modifications (*ContextRefinement* or *ContextCoarsening*) add an acquaintance relationship to a *different* participant *q*, we get a different evolution conflict *EC3: Double source conflict*.

Remark. In the case of an *Operation Invocation Conflict*, we obtained the same conflicts as here, except for *EC2: Double reachability conflict*. Indeed, this conflict does not occur for *«invocation»*-edges because there can be only one *«invocation»*-edge between any two *«operation»*-nodes, and this edge always has an empty label.

The following situation is an example of a domain-specific applicability conflict that does not correspond to a domain-independent conflict. Instead, it is detected by a breach of a domain-specific well-formedness constraint.

Dangling Acquaintance Conflict

Occurs when: A *ContextCoarsening* removes an acquaintance relationship *a* from a participant *p* to participant *q*, while a *ParticipantRefinement* adds an invocation (or a *ParticipantCoarsening* removes an invocation) of operation *p.m* to operation *q.n*.

Corresponds to: Using our formalism, this situation arises because one reuser performs a *Coarsening(a,p,q,«acquaintance»)*, while a different reuser performs a *Refinement(ε,p.m,q.n,«invocation»)*. In our formalism, both contract types are parallelly independent and can be serialised. The result is a graph that contains an *«invocation»*-edge from *p.m* to *q.n*, while the *«acquaintance»*-edge from *p* to *q* has been removed. Consequently, well-formedness constraint C_1 is invalidated, which leads to a *domain-specific applicability conflict*.

Remarks. The other situation, namely *ContextCoarsening* versus *ParticipantCoarsening*, only leads to an invalidation of the well-formedness constraint C_1 if the acquaintance has more than one invocation defined over it.

The next three conflicts are real behavioural conflicts that arise because the addition or removal of operation invocations leads to unpredictable behaviour.

Operation Capture Conflict

Occurs when:
1. A *ParticipantRefinement* adds an invocation to *p.m*, while a different *ParticipantRefinement* (respectively *ParticipantCoarsening*) adds (respectively removes) an invocation from *p.m* to a different operation *q.n*.
2. A *ParticipantRefinement* adds an invocation to *p.m*, while a *ParticipantAbstraction (*respectively *ParticipantConcretisation*) makes this operation *p.m* abstract (respectively concrete).

Corresponds to: The first situation corresponds to *EC1: Reachability conflict*, while the second situation corresponds to *EC6: Inconsistent target conflict*.

Inconsistent Operations Conflict

Occurs when:
1. A *ParticipantCoarsening* removes an invocation that refers to *p.m*, while a *ParticipantRefinement* (respectively *ParticipantCoarsening*) adds (respectively removes) an invocation from *p.m* to a different operation *q.n*.
2. A *ParticipantCoarsening* removes an invocation that refers to *p.m*, while a *ParticipantAbstraction* (respectively *ParticipantConcretisation*) makes this operation *p.m* abstract (respectively concrete).

Corresponds to: We get the same conflicts here as for the previous *Operation Capture Conflict*.

Remark. Our underlying formalism does not make a distinction between *Operation Capture* conflicts and *Inconsistent Operations* because of the uniform way in which we have dealt with *Refinements* and *Coarsenings*. If a distinction is nevertheless desired, the domain-independent evolution conflicts need to be split up in different cases, depending on whether a *Coarsening* or a *Refinement* is used.

Unanticipated Recursion Conflict

Occurs when:
A *ParticipantRefinement* adds an invocation of operation *p.m* to operation *q.n*, while a different *ParticipantRefinement* adds an invocation of operation *q.n* to operation *p.m*. As a result, a cycle is introduced that might lead to an infinitely recursive loop.

Corresponds to: *EC5: Cycle introduction conflict.*

Remark. The same situation is *not* considered to be a conflict if it occurs at the level of acquaintances. Cyclic acquaintance relationships are allowed, as long as they do *not* give rise to unanticipated cyclic operation invocations.

The last three conflicts have to do with making an operation or participant concrete.

Annotation Conflict

Occurs when:
1. A *ParticipantConcretisation* makes an operation *p.m* concrete (by filling in its implementation), while a different *ParticipantConcretisation* does the same (thereby possibly giving a different implementation to *p.m*).
2. An annotation conflict also occurs when we perform two *ParticipantAbstractions* of the same operation *p.m*.

Corresponds to: *AC7: Double node retyping conflict*. Indeed, a *ParticipantConcretisation* (or *ParticipantAbstraction*) corresponds to a *NodeRetyping*, and when we change the type of the same node twice, we obtain a double node retyping conflict.

Mixed Operation Interface Conflict

Occurs when: A *ParticipantConcretisation* makes an operation *p.m* concrete (by filling in its implementation), while a *ParticipantRefinement* (respectively *ParticipantCoarsening*) adds (respectively removes) an invocation from *p.m* to some operation.

Corresponds to: *EC7: Inconsistent source conflict*. Indeed, a *ParticipantConcretisation* corresponds to a *NodeRetyping*, and the *ParticipantRefinement* corresponds to a *Refinement* (respectively *Coarsening*) with the same source node.

Remark. If we would use a *ParticipantAbstraction* instead of a *ParticipantConcretisation*, our formalism would still detect a potential conflict, while this is not considered as a conflict in [Lucas97].

Incomplete Implementation Conflict

Occurs when:

1. A *ContextConcretisation* makes a participant *p* concrete, while a *ParticipantAbstraction* makes a concrete operation *m* in *p* abstract.
2. A *ContextConcretisation* makes a participant *p* concrete, while a *ParticipantExtension* introduces a new abstract operation *m* in *p*.

Corresponds to: Using our formalism, situation 1 would arise when one reuser performed a *NodeRetyping(p,«abs-part»,«conc-part»)*, while a second reuser performed a *NodeRetyping(p.m,«conc-op»,«abs-op»)*. In our formalism, both contract types are parallelly independent and can be serialised. The result is a graph that contains a *«conc-part»*-node with a nested *«abs-op»*-node. Consequently, well-formedness constraint C_2 is invalidated, which leads to a *domain-specific applicability conflict*. The same applicability conflict would also arise when the second reuser would perform an *Extension(p.m,«abs-op»)* with a new operation *m*.

Remark. Even if the well-formedness constraint C_2 would not be present, a conflict would still be detected, but it would be a domain-independent evolution conflict. More specifically, an *EC8: Double node modification conflict* would occur.

VI 2.6 CONCLUSIONS

By taking a close look at the conflicts in the previous section, we can conclude *that our domain-independent reuse contract formalism is a generalisation of [Lucas97]*, since all of the conflicts identified there correspond to an applicability or evolution conflict here. Only a *Dangling Acquaintance Conflict* did not correspond to a domain-independent conflict. Instead, it was detected by a breach of the domain-specific well-formedness constraint C_1. The *Incomplete Implementation Conflict* was another domain-specific applicability conflict, that is detected by a breach of well-formedness constraint C_2. Alternatively, it could also have been detected by the domain-independent evolution conflict *EC8: Double node modification conflict*. This allows us to conclude that *domain-specific well-formedness constraints allow us to find new domain-specific applicability conflicts, or to transform particular domain-independent evolution conflicts into domain-specific applicability conflicts.*

Our reuse contract formalism is not only a generalisation and formalisation of [Lucas97], but also an extension. All applicability conflicts defined in this dissertation correspond to at least one domain-specific conflict in [Lucas97], except for *AC8: Double edge retyping conflict* and *AC10: Undefined edge retyping conflict* that were not needed. The reason why we didn't need these applicability conflicts is obvious: they can only occur in the presence of an *EdgeRetyping*, while none of the primitive modifications defined in [Lucas97] corresponds to an *EdgeRetyping* in our formalism. This is inherently due to the simplicity of the model of class collaborations. As another example, the primitive contract types *Promotion* and *Demotion* introduced in section V 4.2.2 were not needed to deal with class collaborations.

An analogous reasoning can be made for the evolution conflicts: all domain-independent evolution conflicts defined in this dissertation correspond to at least one domain-specific conflict in [Lucas97], except *EC4: Double target conflict* and the nesting conflicts *EC7': Inconsistent source conflict* and *EC6': Inconsistent target conflict.*

We can also conclude that our domain-independent conflicts are finer grained than the domain-specific ones of [Lucas97]. For example, an *Acquaintance Relationship Conflict* coincides with either an *AC5: Duplicate edge conflict*, an *AC6: Double coarsening conflict*, an *EC2: Double reachability* *conflict* or an *EC3: Double source conflict*, depending on the exact form of the involved *ContextRefinements* and *ContextCoarsenings*.

On the other hand, the conflicts *Operation Capture* and *Inconsistent Operations* cannot be distinguished using our formal foundation because of the uniform way in which we dealt with *Refinement* and *Coarsening*. If necessary, however, we can easily fine-tune or domain-independent evolution conflicts so that a distinction between both cases can be made.

Finally, some domain-independent evolution conflicts are not relevant in the domain-specific context, and may consequently be ignored. For example, the domain-independent conflict *EC5: Cycle introduction conflict* only gives rise to a domain-specific *Unanticipated Recursion Conflict* at the level of operation invocations, but not at the level of acquaintance relationships. Similarly, *ParticipantAbstraction* versus *ParticipantRefinement* can lead to a domain-independent *EC7: Inconsistent source conflict* while it is not considered to be a conflict in [Lucas97].

VI 2.7 Experiments

Using our PROLOG implementation of the underlying domain-independent formalism, it was fairly easy to define a domain-specific customisation on top of it. This customisation involved:

- *A translation from domain-specific contract types to domain-independent ones*. Because the names of the domain-independent contract types are somewhat technical (e.g., *NodeRetyping*), it is necessary to give a name which is more meaningful in the domain where the evolution will take place (e.g., *Concretisation*).
- *A translation from domain-specific conflicts to domain-independent ones*. By making use of domain-specific information, more meaningful names can be given to the different evolution conflicts. For example, *Cycle Introduction* is translated into *Unanticipated Recursion*. The translation is not necessarily one-to-one. Several domain-independent conflicts may be combined in a single domain-specific conflict, or a single domain-independent conflict may give rise to more than one domain-specific conflict. The latter is for example the case when a domain-independent conflict is split up in several cases, based on the type of the nodes or edges that are involved.
- *The definition of a domain-specific node type hierarchy and edge type hierarchy.* This can be done based on the kinds of elements and relationships that are used in the domain.
- *The definition of domain-specific well-formedness constraints in a type graph.* In our PROLOG implementation, all constraints are specified as PROLOG rules.
- *The specification of a filter to ignore particular domain-independent evolution conflicts in the specific domain.* Usually, the domain-independent evolution conflicts only give rise to problems for particular domain-specific types of nodes and edges. Moreover, this can differ based on the kind of evolution conflict that is considered. By specifying for each evolution conflict in which cases it should be detected or ignored, we are able to deal with undesired interactions in a more precise way.

VI 2.8 Possible Extensions

The formalism of class collaborations as described in [Lucas97], is too primitive to be practical. Several attempts have already been made to increase its expressiveness, while other attempts are currently going on. Although it is not our intention to discuss this in detail, we will nevertheless give a brief overview of some interesting extensions.

- Besides making a distinction between *abstract* and *concrete* operations and participants, we could also consider other kinds of annotations such as *static*, *public*, *private*, *protected*, *final*, etc. Obviously, each of these annotations will have a different effect on the possible evolution conflicts. Part of this work has already been performed in [Cornelis97].
- As with UML collaboration diagrams, an explicit distinction should be made between the class level, where the classes (participants) and their interfaces (operations) are described, and the object level, where the message interactions (invocations) between different objects (instances of participants) are expressed. This specific extension of [Lucas97] is described in more detail in [Mens&al99a].

- The acquaintance relationship between participants is too restrictive. In practice, we want to be able to make a distinction between different kinds of acquaintances (or associations). For example, in UML, a distinction is made between ordinary associations and aggregations (part-whole relationships), between local and global associations, between temporary and persistent associations, etc. While all these kinds of associations can easily be distinguished by adding different edge types, the interesting part is how this will influence the evolution conflicts that can arise.
- Another impractical restriction is that the approach in [Lucas97] assumes that the so-called “Law of Demeter” should be respected. Basically, this means that cascaded operation invocations are not allowed. In practice, however, this law is seldom respected. Therefore, direct support for cascaded operation invocations should be added.
- The specialisation clauses of operations, denoting the invocations made by each operation, are currently too restrictive. They do not allow for conditional invocations, iteration or ordering of invocations. In order to address these shortcomings, we could extend the specialisation clauses to a variant of regular expressions, but it is far from trivial what the impact of this will be on the possible evolution conflicts.
- Other extensions that should be dealt with are the use of *attributes* (instance variables) as well as an *inheritance mechanism*. This will be done in section VI . 3 , where the evolution of class diagrams is discussed. Note however that this extension does *not* involve behavioural aspects such as self sends (with late binding) and super sends. This specific extension remains future work.
- A final important issue is typing. How can reuse contracts be integrated in a typed language? What is the impact of typing on the evolution conflicts? While this is certainly not a trivial problem, we are currently doing some work in solving these questions.

VI . 3 UML Class Diagrams

As a second customisation of our formal framework, we propose to add support for evolution to UML. Indeed, although it is a “de facto” standard notation for expressing object-oriented analyses and designs, it contains almost no support for dealing with their evolution.

We will focus ourselves on UML class diagrams here. The reason why we have chosen for class diagrams instead of one of the many other kinds of UML diagrams, is that it is the most widespread and most commonly used diagrammatic design notation in the object-oriented community.

VI 3.1 UML

VI 3.1.1 Introduction

UML [OMG97a, OMG97b] has been accepted by the Object Management Group (OMG) as the industry standard for modelling object-oriented software systems. As a result, UML is generally available, well-known, widespread, and there is a lot of tool support. UML offers a wide range of different diagrams in a unified notation. These diagrams can be used in different phases of the software life-cycle. UML can be used for requirements capture, analysis and design, all with a similar notation. UML is also an open language. It contains built-in *extension mechanisms* with which the functionality and behaviour of existing UML modelling elements can be changed. Alternatively, since the semantics of UML is defined using a *metamodel*, one can also change this semantics by directly editing the metamodel, at the risk of losing compatibility and portability.

Although the wide acceptance of UML as an object-oriented analysis and design notation, it does not provide adequate support for dealing with reusable and evolvable software artifacts. This can be concluded from the following paragraph extracted from [OMG97b]:

> *Whenever the supplier element of a dependency changes, the client element is potentially invalidated. After such invalidation, a check should be performed followed by possibly changes to the derived client element. Such a check could be performed after which action can be taken to change the derived element to validate it again. The semantics of this validation and change is outside the scope of UML.*

Because of this, there is a clear need for adding support for evolution to UML. We illustrate how this can be achieved by customising the domain-independent reuse contract framework to a specific kind of UML diagrams, namely *class diagrams*. We will later sketch how the ideas can be extended to other kinds of diagrams.

VI 3.1.2 Labelled Typed Graph Notation

Although we did not explicitly mention it before, the notation we use in this dissertation for representing labelled typed graphs is borrowed from UML. In UML, each model element has a name as well as an optional stereotype, and a (possibly empty) set of constraints and tagged values. The name corresponds to the label of a node or edge, the stereotype (depicted between guillemets *«…»*) corresponds to the type of a node or edge, and the constraint (depicted between curly braces *{…}*) corresponds to a node constraint or edge constraint. The only difference here is that constraints in UML are usually expressed in the Object Constraint Language OCL [OMG97d].

The graphical notation for nodes and edges themselves is also consistent with UML. Nodes are represented by a rectangle, which corresponds to the (collapsed) class notation of UML. Edges are depicted by a plain line with an arrow, corresponding to the (unidirectional) association notation of UML.

VI 3.1.3 UML Metamodel

The UML semantics is described using a *metamodel* that consists of three views:

- The *abstract syntax*, which is expressed in a subset of the UML notation.
- *Well-formedness rules* specify when an instance of a particular language construct is meaningful. These rules are expressed partly in natural language, and partly as invariants in OCL. Multiplicity

and ordering constraints do not have to be described in these well-formedness rules, since they are already present in the abstract syntax.

- The (dynamic) *semantics* defines the meaning of a well-formed language construct. It is (unfortunately) described primarily in natural language.

The UML metamodel is defined in a metacircular way, using a subset of UML notation and semantics to specify itself. In this way, the UML metamodel bootstraps itself in a manner similar to how an interpreter is used to compile itself. The UML metamodel consists of a four-layered structure. In this layered structure, each layer is an instance of a higher layer. This is even true for the highest layer, namely the meta-metamodel, which is actually an instance of itself. However, we will not go in detail on this. Graphically, the structure is presented in Figure 39.

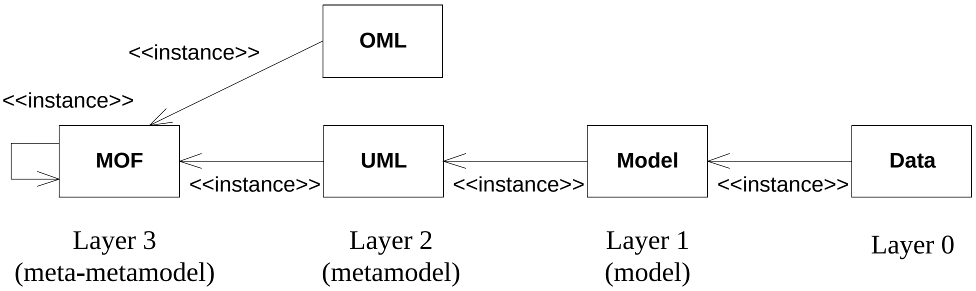


Figure 39: Four-layered metamodel structure

- The *meta-metamodel* defines the language for specifying metamodels. It provides the infrastructure for a metamodelling architecture. Currently, the UML metamodel is specified by using a meta-metamodel which is called the Meta Object Facility (MOF) [OMG97c]. One should be aware that MOF is not the only meta-metamodel available. For example, in the OPEN consortium, the Common Object Methodology Metamodel Architecture (COMMA) is preferred [Henderson-Sellers&Bulthuis98]. Regardless of the meta-metamodel one chooses, there are several advantages. First of all, it simplifies transition from one modelling language (or metamodel) to another, as long as both are expressed in the same meta-metamodel. For example, in Figure 39 we have depicted both UML and OML [Firesmith&al97] as instances of the same meta-metamodel. A meta-metamodel is also useful for comparing different modelling languages, since it reveals similarities and differences which go unnoticed often in informal comparisons. A third advantage of a meta-metamodel is that it allows one to add new features to the modelling language if these features do not yet exist.
- The *UML metamodel* itself is an instance of the meta-metamodel, and actually defines the UML language.
- Using the UML metamodel, different *UML models* can be instantiated. Each of these models can be seen as a language that describes an information domain.
- *Data* or *user objects* are instances of a *UML model*. They define a specific information domain.

VI 3.2 CLASS DIAGRAMS

VI 3.2.1 Informal Discussion

Class diagrams show the static structure of an object-oriented system, such as the classes that exist in the system, their features (operations and attributes), and their relationships to other classes (by means of association, aggregation, generalisation, etc…). UML class diagrams may also contain packages for abstraction purposes, to reduce the complexity of a given diagram. Class diagrams may also contain interfaces [Canning&al89], as can be found in the Java programming language.

Class diagrams can be (and have been) used for different purposes.

- Class diagrams can be used for *data modelling*, and essentially provide an extended Entity Relationship notation [Chen76]. Classes correspond to relations in a database, instance variables (or attributes) correspond to columns in a relation, etc. Instead of using the term class diagram, we speak of an *object-oriented database schema* [Banerjee&al87, Kim&al89, Barbedette91].

- Classes are very frequently used at design and implementation level. Associations are used to represent inter-object communications. Implementation details such as operations, navigation and visibility indicators (public/protected/private) can be specified as well. The design of a software system is usually described by means of *class diagrams and collaboration diagrams*, sometimes augmented with *state diagrams*. While we have already dealt with collaboration diagrams (or at least a variant thereof) in section VI . 2 , we will focus on class diagrams in this section.

When comparing both uses of class diagrams, it is obvious that the focus in the former approach lies on modelling, modifying and migrating data (attributes), with less emphasis placed on operations. Nevertheless, there are some facilities to detect when operations have been invalidated because the data they reference has been altered due to a schema change. It should also be noted that a class diagram used for data modelling cannot be translated directly and extended to a class diagram used for expressing inter-object communications. For example, in the data modelling approach associations usually describe data dependencies, and database normalisation techniques can be applied to these dependencies. On a design level however, associations are used to represent inter-object communications, which sometimes gives rise to connections that violate the normalisations from the data-modelling perspective [Simons&Graham98]. In [Firesmith&al97] it is recognised that data modelling and class modelling are different activities. For this reason class diagrams should be less influenced by data modelling techniques. A similar argument is given in [Halpin98], where Terry Halpin advocates to use ORM diagrams (Object Role Modelling) instead of class diagrams for the purpose of data modelling. For these reasons, we will only focus on class diagrams used for modelling designs rather than data.

VI 3.2.2 Example

A class diagram basically consists of *classes*, which can contain *operations* and *attributes*, and between which *generalisation* and *association* relationships can be defined. Some of these association relationships can be composite associations or *aggregation* relationships. In Figure 40, an example is given of a real-world class diagram. It is a small but essential extract of the design of an object-oriented framework in C++ consisting of about 600 classes. The framework deals with dossier management for the Belgian Courts of Appeal.

The *generalisation* relationship is represented by a line ending in a white triangle. It models the inheritance or incremental modification mechanism. Each class is subdivided in three compartments: the name of the class, the *attributes*, and the *operations*. Compartments that are empty or irrelevant may be hidden. Operations and attributes in a superclass (parent) automatically get inherited by all subclasses (descendants). When an object is created as an instance of a class in the class diagram, it automatically understands all operations defined in the class or any of its superclasses (ancestors). Moreover, it provides a value for all attributes defined in the class and its superclasses.

Association relationships can be put between classes. They end in an arrow if they are unidirectional, which is the case for all the associations in the example. Composite (or aggregate) associations can be recognised by a filled diamond at their start. Usually, multiplicity information as well as association role names are also mentioned on the association, but we have decided to leave them out in the example in order not to make the picture overly complex.

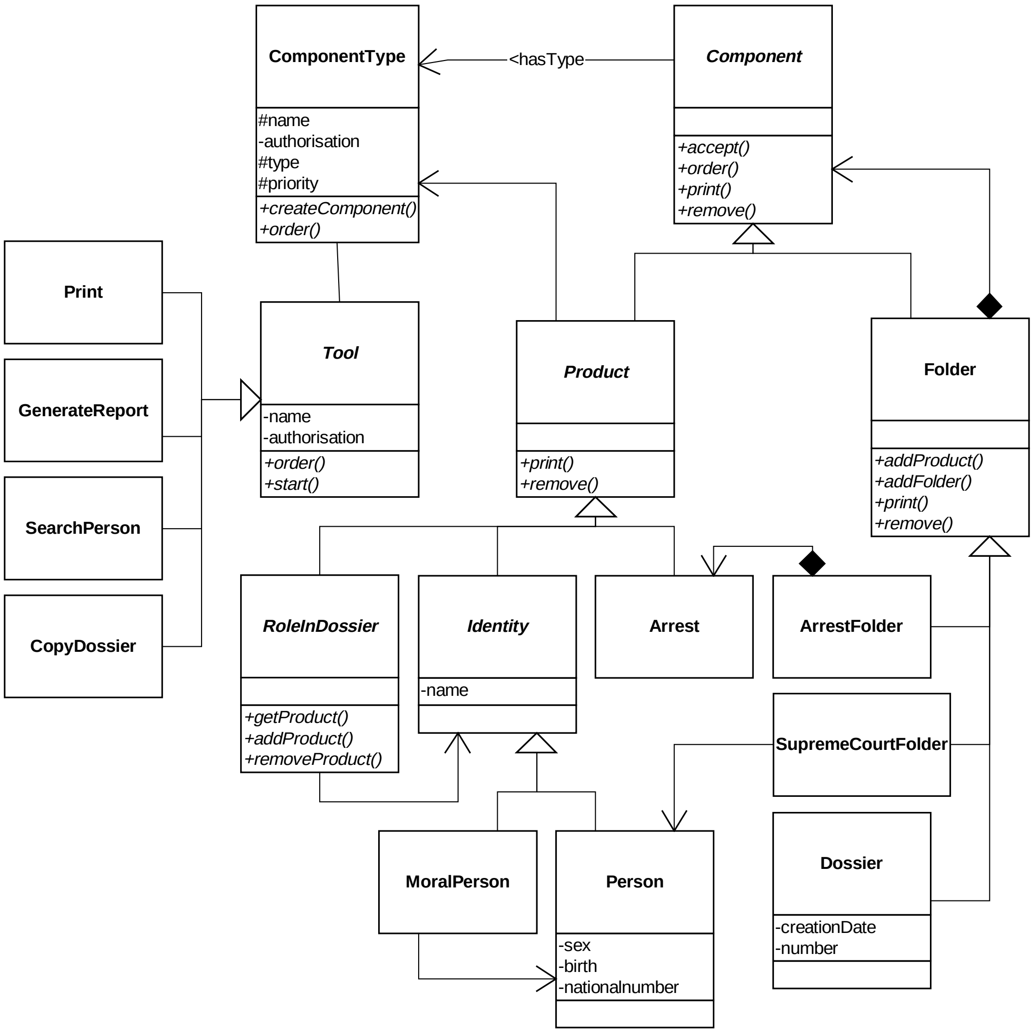


Figure 40: Extract from a large class diagram

The main difference between operations and attributes is that different instances of the same class have exactly the same implementation for each of their operations, but can have different values for their attributes. In other words, the operations are shared by all instances, while the attributes are not. Attributes or operations in a class are automatically inherited by its subclasses (except when they are “overridden” or redefined). This requires a *lookup mechanism* that traverses all superclasses of a given class to see if a particular operation or attribute is understood.

In Figure 41, our formal representation of class diagrams is shown on a subset of Figure 40. We have three *«class»*-nodes *Component*, *Folder* and *ComponentType*, each having a number of *«operation»*-nodes and *«attribute»*-nodes nested inside. The classes are connected by means of *«association»*-edges and *«generalisation»*-edges. The latter ones represent inheritance relationships, in this specific case between *Folder* and *Component*. *Folder* understands all operations and attributes nested in itself and in *Component*. Moreover, the operations *print* and *remove* are redefined or *overridden*, because they are explicitly mentioned in *Folder* as well as *Component*. The technique of overriding operations in subclasses is very powerful, since it allows creation of polymorphic operations, i.e., operations with the same name but a different implementation.

Associations between classes can be unidirectional or bidirectional. In the latter case, they will be represented by two different *«association»*-edges (in the opposite direction) in the graph.

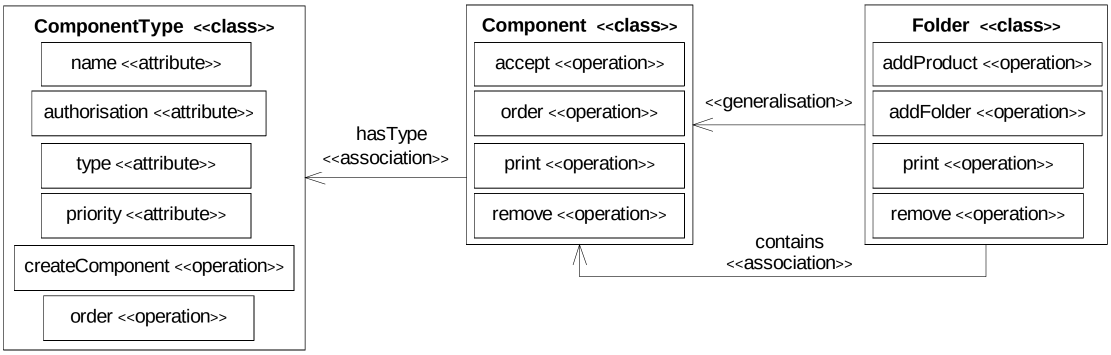


Figure 41: Graph representation of a class diagram

VI 3.2.3 Class Diagrams in the UML Metamodel

In order to add support for evolution to UML class diagrams, we need to map all the elements in such a class diagram to nodes and edges of a graph. Therefore, we first need to determine which kinds of node types and edge types are needed. In other words, we need to map the different kinds of entities in a class diagram to different kinds of node types or edge types in a graph.

The fact that the UML semantics is defined in terms of a metamodel makes it much easier to identify node types, edge types and their type constraints for all the different constructs in a class diagram. Intuitively, each metaclass in the metamodel that formally describes an entity or relationship in an UML class diagram will correspond to a node type or edge type in the underlying graph formalism. In order to know which metaclasses need to be mapped on node types, and which metaclasses need to be mapped on edge types, we will take a closer look at the UML metamodel for class diagrams.

In Figure 42, a simplified subset of the UML 1.3 metamodel is presented.[5] Only those metaclasses that are directly relevant to class diagrams are mentioned. Other intermediary metaclasses, such as *NameSpace*, *GeneralizableElement*, *AssociationClass*, *AssociationEnd*, *StructuralFeature* and *BehaviouralFeature* were purposefully omitted. Also, we will not deal with all aspects of UML class diagrams for the sake of the presentation. For example we will not consider *Methods*, *Dependencies* and *DataTypes* here.

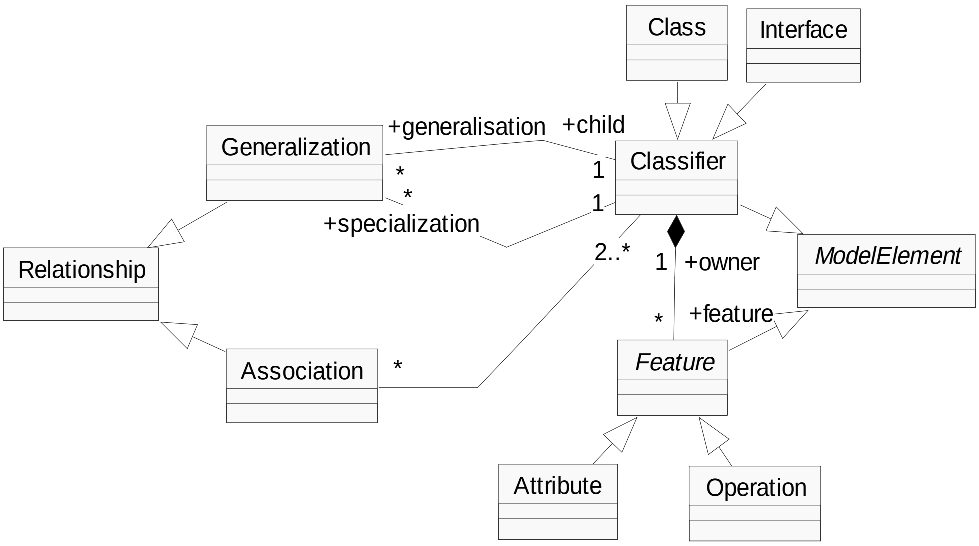


Figure 42: Stripped-down subset of the UML metamodel

[5] Note that the UML metamodel itself is also expressed using class diagram notation.

We can immediately distinguish a special metaclass called *Relationship* in the metamodel. It represents any kind of relationship that is used to link other model elements together.[6] Examples of this are *Generalisations*, *Associations* and *Dependencies*.

Besides *Class*, the metamodel also mentions *Interface*. We didn't encounter this concept in our introductory example. Interfaces are similar to classes, except that they only contain operations (no attributes), and they do not have an implementation associated to these operations. They can be used as a more abstract view on classes, or as a typing mechanism.

VI 3.3 TYPE CONSTRAINTS

In this section we discuss the type constraints required to formally represent UML class diagrams in terms of labelled typed graphs. We will make use of the UML metamodel to extract this information.

VI 3.3.1 Type Partial Orders

Because every kind of relationship in UML is defined as (an instance of) a subclass of the metaclass *Relationship*, it is natural to consider all these subclasses (such as *Association* and *Generalisation*) as edge types. All other metaclasses (which are direct or indirect specialisations of *ModelElement*) will be mapped on node types (e.g., *Classifier* and *Feature*). In this way, we get a node type partial order and edge type partial order that exactly mimic the corresponding generalisation hierarchies of Figure 42. Both partial orders are shown in Figure 43.

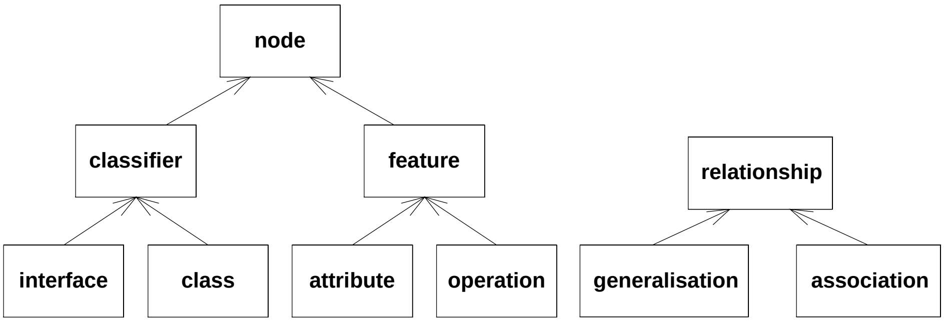


Figure 43: Type partial orders for UML class diagrams

From the node type partial order we see that *interface* and *class* are similar concepts, since they are both a subtype of *classifier*, which captures the similarities of both. Likewise, *feature* is used to capture the similarities of both *attribute* and *operation*.

VI 3.3.2 Type Graph

Having defined the necessary node types and edge types, we can add additional constraints to these types. Again, these constraints can be extracted immediately from the UML metamodel. Any association in the metamodel that connects a subclass of *Relationship* to one or more subclasses of *ModelElement* will correspond to an edge in the type graph. For example, the type graph contains a *generalisation* edge with *classifier* as source and target node, because there are two associations (a child and a parent) from *Generalization* to *Classifier* in the UML metamodel. The complete type graph is shown in Figure 44.

[6] Strictly speaking, *Relationship* is also a specialisation of *ModelElement*, but we will act as if this were not the case to avoid unnecessary technical difficulties that will only clutter the presentation.

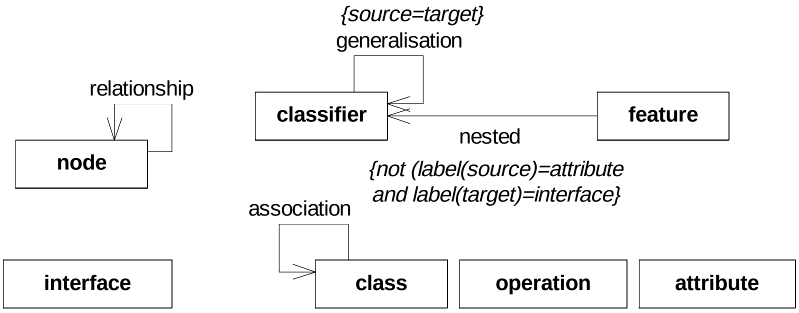


Figure 44: Type graph for class diagrams

When comparing Figure 44 with Figure 42, some important remarks need to be made.

- Subclasses of *Relationship* are represented as edges instead of nodes in the type graph.
- The composite association relationship (a filled diamond) between *Classifier* and *Feature* in the UML metamodel of Figure 42 is translated into a *nested* edge in the type graph. Keep in mind that *nested* is not a “real” edge type, but instead imposes constraints on the nesting hierarchy of graphs.
- Multiplicity constraints could be added to Figure 44 to express additional requirements like whether multiple inheritance is allowed or not. While it *is* allowed in the UML, we will not deal with any multiple inheritance aspects here for the sake of the presentation.
- All constraints specified on nodes *classifier* and *feature* in the type graph are automatically inherited by *interface*, *class*, *attribute* and *operation*, which are subtypes according to the type partial order of Figure 43.
- In the UML metamodel, an *Association* was connected to the *Classifier* metaclass. We have put the *association* edge on *class* (instead of *classifier*), because there is an additional OCL constraint in the UML semantics that specifies that an *Interface* cannot have association relationships. Additionally, and *Association* was allowed to correspond to more than two *Classifiers* in the metamodel. Such arbitrary n-ary associations cannot be expressed in our current formalism, since each edge can have only one source and one target node. However, if we would extend our formalism to deal with *hyperedges* instead of edges, this case would be solved.

VI 3.3.3 Additional Type Constraints

As always, there are some constraints that cannot be expressed immediately in the type graph, or even in the UML metadiagram. In UML, the approach taken is to express all these extra constraints in OCL. In our approach, the constraints are expressed in mathematical notation.

The first constraint puts an additional restriction on *«interface»*-nodes: they are only allowed to have *«operation»*-nodes nested in them. In other words, *«attribute»*-nodes cannot be nested in *«interface»*-nodes. This constraint was also expressed as a textual constraint in Figure 44: *{not (label(source)=attribute and label(target)=interface)}*.

Constraint C_1: *«interface»*-node:

$\forall (v,\text{«interface»}) \in V$: if $w \in V$ with $(w,v) \in \textit{nested}$ then $\textit{type}(w) = \text{«operation»}$

Constraint C_2 specifies that the source and target of a *«generalisation»*-edge must have the same type. In other words, *«generalisation»*-edges are only allowed between two *«class»*-nodes or between two *«interface»*-nodes, but not between a *«class»*-node and an *«interface»*-node. This constraint was expressed textually as *{source=target}* in the type graph of Figure 44. By convention, we will attach an empty label ε to each *«generalisation»*-edge.

Constraint C_2: *«generalisation»*-edge:

$\forall (\varepsilon,u,v,\text{«generalisation»}) \in E$: $\textit{type}(u)=\textit{type}(v)$

While, in general, the triple (e,u,v) of edge label, source node label and target node label is required to be unique in an arbitrary graph G, constraint C_3 states a stronger requirement. In the case of *«association»*-edges, the pair (e,u) of edge label and source node label should be unique. In other words, the same source node should not be the source of different *«association»*-edges with the same label.

Constraint C_3: *«association»*-edge:

$\forall(e,u,v,\text{«association»})\in E$: $\nexists w\neq v$ such that $(e,u,w,\text{«association»})\in E$

We do not need an explicit constraint to express that the labels of operations and attributes in the same class should be unique, since the injectivity constraint on nested graphs automatically requires labels of nodes that are nested in the same node to be unique.

Constraint C_4 states that each *«class»*-node is associated to itself by means of an *«association»*-node labelled *self*. Because of the injectivity constraint on edges of a graph, this association is unique for each *«class»*-node.

Constraint C_4: *«class»*-node:

$\forall(v,\text{«class»})\in V$: $\exists(self,v,v,\text{«association»})\in E$

The next three constraints deal with *«generalisation»*-edges, and are considerably more complex than what we have encountered so far. They can only be defined by using the *transitive closure of «generalisation»-edges*. More precisely, the constraints on an arbitrary graph G need to be defined in terms of a graph $\boldsymbol{H = (generalisation(G))^{+}}$, which is obtained by first taking the *«generalisation»*-spanning subgraph of G (Definition 17), and then taking its transitive closure (Definition 10) to deal with paths of *«generalisation»*-edges of arbitrary length. Additionally, we require H to have the same nesting hierarchy as G, i.e. $nested_H = nested_G$.

The first of the transitive closure constraints specifies that the inheritance hierarchy should not contain cycles. It can be a multiple inheritance hierarchy, though.

Constraint C_5: *«generalisation»*-edge:

$\forall(\varepsilon,v,w) \in E_H$: $(\varepsilon,w,v) \notin E_H$

The last two constraints say something about overriding of features. Again, we have to take the transitive closure into account. If we deal with *«generalisation»*-edges between *«class»*-nodes, attribute overriding is not allowed (in our approach). If an attribute is defined in a class, it cannot be defined in any of its descendants (subclasses). Similary, operation overriding is not allowed between *«interface»*-nodes. If an operation is defined in an interface, it cannot be defined in any of its descendants.

Constraint C_6: *«generalisation»*-edge:

$\forall(\varepsilon,v,w) \in E_H$: $\forall(a,\text{«attribute»}), (b,\text{«attribute»}) \in V_H$:

if $(a,v) \in nested_H$ and $(b,w) \in nested_H$ then $label(a) \neq label(b)$

Constraint C_7: *«generalisation»*-edge:

$\forall(\varepsilon,v,w) \in E_H$: $\forall(o,\text{«operation»}), (p,\text{«operation»}) \in V_H$:

if $(o,v)\in nested_H$ and $(p,w)\in nested_H$ and $type(v)=type(w)=\text{«interface»}$ then $label(o)\neq label(p)$

When specifying class diagrams for data modelling, we have similar constraints. They are referred to as *schema invariants*, because they assure that the well-formedness constraints are preserved after a change has been made to an object-oriented database schema. In other words, a schema is not allowed to be modified in an inconsistent way. In [Kim&al89], constraint C_5 is referred to as the *class lattice invariant*. Constraint C_3 is part of the *distinct name invariant*, which specifies that all elements known to a class (attributes, operations and associations) must have distinct names. There are some other invariants as well, but we will not discuss them here.

VI 3.4 Domain-Specific Contract Types

VI 3.4.1 Schema Transformations

In order to specify the different ways in which a class diagram can be modified, we rely on existing research in object-oriented database schema evolution [Banerjee&al87, Kim&al89, Barbedette91]. In object-oriented databases, schema transformations are used to evolve an object-oriented database schema. This is for example the case in ORION [Kim&al89], a prototype object-oriented database system built in Common LISP. In this system, [Banerjee&al87] established a taxonomy of 19 different schema transformations. We have used these transformations as a guideline for defining our own domain-specific contract types. Obviously, we will need to make some changes to the proposed transformations, to deal with the differences with our approach. We also need some new transformations, for example to deal with interfaces.

Although some of the transformations will correspond to primitive contract types while others are composite, we will discuss them all in this same subsection.

AddOperation(C.o)

Description. Add a new operation *o* to an existing classifier *C*.
Corresponds to: *Extension(C.o,«operation»)*
Remark. This definition holds for both *«class»*-nodes and *«interface»*-nodes because *«operation»* can be nested in any *«classifier»*-node according to the type graph. Once an operation *o* is added to a classifier *C*, it automatically gets inherited by all the descendants of *C* thanks to the operation lookup mechanism.
If *C* is an *«interface»*-node, we have to check the additional restriction that *o* is not defined in any ancestor or descendant of *C*. Otherwise, constraint C_6 would be breached, since operations in an interface cannot be overridden.
If *C* is a *«class»*-node, no additional restrictions are needed, since overriding of operations is allowed.

DropOperation(C.o)

Description. Remove an existing operation *o* from an existing classifier *C*.
Corresponds to: *Cancellation(C.o,«operation»)*
Remark. Again, the definition holds for both *«class»*-nodes and *«interface»*-nodes. An operation can only be removed from the classifier where it is defined. In other words, an inherited operation can only be removed indirectly, by removing it in the parent where it is defined.

If we want to remove an operation *o* entirely from a class hierarchy, it is possible that we need to apply *DropOperation* repeatedly, since *o* might have been overridden in several subclasses, and all these definitions must be removed as well. Obviously, this can be achieved by making use of a (monotonous) composite contract type which is composed of several *DropOperations*.

RenameOperation(C,o,p)

Description. Change the name of an operation *o* in a classifier *C* to *p*.
Corresponds to: *RelabelNode(C.o,«operation»,C.p)*
Remark. Obviously, renaming can only take place in the classifier where the operation is defined, not in classifiers that inherit the operation from a parent. Also, renaming the operation from *o* to *p* is only allowed if there is not yet an operation *p* defined in *C* or any of its ancestors or descendants. Otherwise, we get a name collision or unintended redefinition of *p*. If *C* is an interface, this even coincides with a breach of constraint C_6.

Analogously to the domain-specific contract types for modifying operations, we can define contract types for modifying (adding, removing or renaming) attributes. This is only possible for classes, since constraint C_1 prohibits attributes to be nested in interfaces.

AddAttribute(C.a)

Description. Add a new attribute *a* to a class *C*.
Corresponds to: *Extension(C.a,«attribute»)*
Remark. Because we do not allow attribute overriding (constraint C_7), this contract type cannot be applied if the attribute *a* already exists in one of the ancestors or descendants of *C*.

DropAttribute(C.a)

Description. Delete an existing attribute *a* from a class *C*.
Corresponds to: *Cancellation(C.a,«attribute»)*
Remark. The attribute *a* can only be deleted from the class in which it is defined. In other words, an inherited attribute can only be removed indirectly, by removing it in the parent where it is defined.

RenameAttribute(C,a,b)

Description. Change the name *a* of an attribute of a class *C* to *b*.
Corresponds to: *RelabelNode(C.a,«attribute»,C.b)*
Remark. The attribute *a* can only be renamed to *b* if *b* itself is not already defined in *C* or any of its ancestors or descendants. Otherwise, we get a name collision or a breach of constraint C_7.

The next domain-specific contract type specifies how a *«generalisation»*-relationship can be added between two existing classifiers.

AddGeneralisation(C,P)

Description. Make a classifier *P* a parent of a different classifier *C* by adding a *«generalisation»*-edge from *C* to *P*. As a result, all operations of *P* become inherited by *C*. In the case of classes, the attributes of *P* become inherited by *C* as well.
Corresponds to: *Refinement(ε,C,P,«generalisation»)*
Remark.
Because of constraint C_5, this modification should only be allowed if the introduction of the new *«generalisation»*-relationship does not introduce any cycles in the inheritance hierarchy.
If *C* is a class, we have to pose the additional restriction that *C* (or any of its descendants) does not have an attribute that already exists in *P* (or any of its ancestors). Otherwise, constraint C_7 would be breached, since overriding of attributes in classes is prohibited.
If *C* is an interface, we have to pose the additional restriction that *C* (or any of its descendants) does not have an operation that already exists in *P* (or any of its ancestors). Otherwise, constraint C_6 would be breached, since overriding of operations in interfaces is prohibited.

The next domain-specific contract type is used to pull a classifier one level up in the inheritance chain (*PullUpClass*). In an analogous way its inverse of pushing a classifier one level down in the inheritance chain can be defined (*PushDownClassifier*).

PullUpClassifier(C, P, Q)

Description. Redirect a *«generalisation»*-relationship between classfier *C* and its parent *P* to a parent *Q* of *P*.
Corresponds to: *RedirectTarget(ε,C,P,«generalisation»,Q)*
Remark.
This modification does not have an effect on other classes that might have *P* as parent. *P* is not removed from the diagram by this contract type.
If *C* is a class, operations in *C* which were redefinitions of operations in *P* are still allowed, although they can lead to problems if their implementations make use of super sends.

Finally, we can introduce some elementary contract types on classifiers: introducing, removing or renaming a classifier.

AddClassifier(C)

Description. Add a new empty classifier *C* to a class diagram.
Corresponds to: *Extension(C,«classifier»)*
Remark. A classifier can only be introduced without *«generalisation»*-edges to existing classifiers. If these relationships are needed, they must be introduced later by means of *AddGeneralisation*. Similarly, operations and attributes have to be added later.

DropLeaf(C)

Description. Remove a classifier *C* from a class diagram, under the restriction that *C* does not have any descendants.
Corresponds to:
*[*Coarsening*(ε,C,P,«generalisation»), DropOperation(C.o_1), …, DropOperation(C.o_n), DropAttribute(C.a_1), …,DropAttribute(C.a_m), Cancellation(C,«classifier»)]* where *P* is the parent of *C*, o_1,…,o_n are all operations defined in *C*, and a_1, …, a_m are all attributes defined in *C*.
Remark. If *C* is an interface, we do not need the primitive contract types *DropAttribute* in the above definition. If *C* does not have a parent *P*, we do not need the *Coarsening* in the above definition.

DropClassifier(C)

Description. Remove a non-leaf classifier *C* from a class diagram. To do this, first make it a leaf by redirecting all *«generalisation»*-relationships from its subclasses C_i to its superclass *P*. Next, remove the leaf classifier C.
Corresponds to: *[PullUpClassifier(C_1,C,P), …,PullUpClassifier(C_n,C,P), DropLeaf(C)]* where C_1, …, C_n are all subclasses of *C*.
Remark. In the presence of multiple inheritance, this definition needs to be extended to deal with multiple parents of *C*. If *C* does *not* have a parent *P*, the definition of *DropClassifier* can be simplified by removing all *PullUpClassifiers*.

RenameClass(C,D)

Description. Change the name of a classifier *C* in a class diagram to *D*.
Corresponds to: *RelabelNode(C,«classifier»,D)*

To introduce, remove or rename *«association»*-edges in a class diagram, we can use the following three domain-specific contract types:

AddAssociation(a,C,D)

Description. Add an association relationship with name *a* between two existing classes *C* and *D*.
Corresponds to: *Refinement(a,C,D,«association»)*
Remark. Because of constraint C_3, we can only add the association if the label *a* is not already used for a different association.

DropAssociation(a,C,D)

Description. Remove an association relationship with name *a* between two existing classes *C* and *D*.
Corresponds to: *Coarsening(a,C,D,«association»)*

RenameAssociation(a,C,D,b)

Description. Change the name of an association relationship between classes *C* and *D*.
Corresponds to: *RelabelEdge(a,C,D,«association»,b)*
Remark. Because of constraint C_3, the renaming is only allowed if label *b* is not yet used for a different association that has the same source node *C*.

Another useful domain-specific contract type on associations allows us to change the source or target of an association relationship between classes:

AssociationRedirection(a,A,B,C)

Description. Change the source or target node of an existing association relationship (with name *a*).
Corresponds to:
RedirectSource(a,A,B,«association»,C) or *RedirectTarget(a,A,B,«association»,C)*
Remark. Again we have to check if the redirection does not breach constraint C_3.

A frequently needed composite contract type is to create a new specialisation *C* of an existing classifier *P*. It can be defined in terms of *AddClassifier* and *AddGeneralisation*. A similar contract type can be defined for dealing with interfaces.

ClassifierSpecialisation(P,C)

Description. Create a new subclass *C* of an existing class *P*.
Corresponds to: *[AddClass(C), AddGeneralisation(C,P)]*

VI 3.4.2 Other Useful Contract Types

In [Banerjee&al87], some other transformations were introduced, mainly to deal with multiple inheritance name collisions. Because we do not deal with multiple inheritance here, we will not use these operations. Note that [Banerjee&al87] is not the only one that gives a taxonomy of schema modifications. For example, [Barbedette91] presents a taxonomy of very similar operations in $LISPO_2$, a persistent object-oriented language consisting of LISP combined with the O_2 object-oriented data model and orthogonal persistence.

In the literature about object-oriented refactorings [Opdyke92, Johnson&Opdyke93, Tokuda&Batory98b], many behaviour-preserving transformations on class diagrams are proposed. Each of them can be defined as a domain-specific composite contract type in our formalism. We will only summarise some interesting refactoring transformations here.

MoveAttribute is a transformation that can move an attribute from a class to a different one. In terms of this transformation, we can also define the more complex ***PullUpAttribute***, which pulls up an attribute to an abstract superclass if it is defined in each its subclasses. An analogous reasoning can be made for the transformations ***MoveOperation*** and ***PullUpOperation***, except that these can also be defined on interfaces instead of classes. In fact *PullUpAttribute* and *PullUpOperation* can be seen as finer-grained versions of *PullUpClassifier*. We can also define their inverses ***PushDownAttribute*** and ***PushDownOperation***.

Other useful behaviour-preserving contract types could be ***ClassToInterface*** and ***InterfaceToClass***, for changing a class (which does not have any attributes) into an interface and vice versa. They can be defined in terms of the primitive contract type *NodeRetyping*.

Because all these refactoring transformations preserve the behaviour of the class diagram in some way, they often give rise to fewer actual evolution conflicts than will be generated by the conflict detection algorithm. As was the case for the domain-independent composite contract type *Factorisation* of section V 3.1 (page 129), it is likely that particular domain-specific evolution conflicts may be ignored for each of the above behaviour-preserving modifications.

VI 3.5 Applicability and Evolution Conflicts

In the context of an industrial case study, in collaboration with an industrial partner, we are currently performing experiments with the above customisation of our domain-independent formalism to UML class diagrams. Below we report on some preliminary results of these experiments. More specifically, we discuss some interesting conflicts that can be detected for evolving class diagrams.

VI 3.5.1 Domain-specific Applicability Conflicts

Because of the extra domain-specific well-formedness constraints needed to deal with the domain of class diagrams, there are obviously a number of *domain-specific applicability conflicts* that can arise.

If we have two classes *C* and *D* that are not connected, different evolvers can decide to add a generalisation relationship in the opposite direction (by using *AddSpecialisation(C,D)* and *AddSpecialisation(D,C)*, respectively). The result of this is that constraint C_5 gets invalidated, because a cycle is introduced in the *«generalisation»*-induced subgraph. The same problem can also occur with cycles of length more than two that can only be detected by taking the transitive closure into account.

Another example of a domain-specific applicability conflict arises when two different evolvers add an association with the same name *a* from the same class *C* to different classes *D* and *E* (by means of *AddAssociation(a,C,D)* and *AddAssociation(a,C,E)*, respectively). The combination of both modifications leads to a breach of constraint C_3.

A breach of constraint C_6 arises when one evolver adds an operation *o* to interface *A*, and a second evolver adds an operation *o* to interface *B*, while *A* is an ancestor of *B* or vice versa. A breach of constraint C_7 arises when one evolver adds an attribute *a* to class *C*, and a second evolver adds an attribute *a* to class *D*, while *C* is an ancestor of *D* or vice versa. Breaches of either constraint C_6 or C_7 can also arise when independent evolvers perform an *AddGeneralisation* and an *AddOperation* (or *AddAttribute*) involving the same classifier (either directly or indirectly).

Finally, the contract types *ClassToInterface* and *InterfaceToClass* can lead to domain-specific applicability conflicts because constraint C_1 or C_2 becomes breached.

VI 3.5.2 Evolution Conflicts

Almost all domain-independent evolution conflicts defined in section IV 4.4 have to do with the introduction (or removal or retyping) of edges. Since we make a distinction between *«association»*-edges and *«generalisation»*-edges, we will first look at evolution conflicts for both types of edges separately, and then look at conflicts involving a combination of both types of edges.

Adding or removing an *«association»*-edge is a fairly conservative modification. All evolution conflicts involving an introduction of two such edges by independent evolvers (like *EC1: Reachability conflict*) may be ignored. An exception is the *EC3: Double source conflict*, which yields a domain-specific evolution conflict when different evolvers add an *«association»*-edge with the same name but a different target-node to the same source node. However, this coincides with a domain-specific applicability conflict, because it leads to a breach of constraint C_3.

Adding or removing a *«generalisation»*-edge is much more intrusive, because of the semantics that is usually associated with it. A *«generalisation»*-edge from *C* to *P* indicates that *C* can be considered as a kind of "specialisation" of *P*, in the sense that its behaviour or functionality is enhanced. Below, a number of evolution conflicts are discussed that arise when independent evolvers both add a *«generalisation»*-edge.

- An *EC5: Cycle introduction conflict* will coincide with a domain-specific applicability conflict because it leads to a breach of constraint C_5.
- An *EC3: Double source conflict* will also lead to a domain-specific applicability conflict if multiple inheritance is not allowed. It it is, a warning should still be generated, since the fact that different evolvers independently choose a different parent for the same class might indicate an inconsistency in the design of the software.
- An *EC2: Double reachability conflict* can only arise if we take the transitive closure of *«generalisation»*-edges into account. It can be an indication of *multiple inheritance name collisions*, since a class is inherited from an other one via two different paths.
- The nested evolution conflict *EC6': Inconsistent target conflict* represents the typical problem of *parent class exchange*. It occurs when a class *P* is modified in some way, while a *«generalisation»*-edge from *C* to *P* is added by an independent evolver. Because this evolver is unaware of the fact that *P* has been modified, this can give rise to undesired interactions in *C*. It can even have an impact on all descendants of *C*, but therefore we need to take the transitive closure into account. Note that the conflict as it is described here is too coarse-grained: a conflict will be detected in many situations where there is no problem. We can deal with the problem of parent class exchange in a more detailed way by specifying the precise way in which *P* is modified, as well as the exact changes that are made by the *«generalisation»* from *C* to *P*. This can be achieved by documenting the *«generalisation»*-edge with reuse contract information, in the way described in section V 5.3. A slight variation of this approach was taken in [Steyaert&al96], where the different kinds of evolution conflicts that can be detected in this way are explained.

There are some interesting evolution conflicts that arise because of an interaction between *«generalisation»*-edges and *«association»*-edges.

- Suppose we have a class *C* which contains a *«generalisation»*-edge to its parent class *P*. If one evolver adds an *«association»*-edge from *P* to a different class *A*, while an other evolver adds an association to *A* from the child class *C*, this is likely to pose a conflict. Indeed, because the association of *P* gets inherited by *C*, it can conflict with the association that was introduced in *C*. This conflict can be detected with our formalism as an instance of *EC2: Double reachability conflict*. Indeed, *A* can be reached from *C* via two paths. One direct association, and one indirect association through the superclass. Note that this conflict can only be detected if we take the second-order closure into account. Similar situations may be imagined where an even higher-order closure is necessary.
- A similar conflict to the one above arises when we have a class *C* which already contains an *«association»*-edge to class *A*. If one evolver decides to add a *«generalisation»*-edge from *C* to class *P*, while an independent evolver decides to introduce an *«association»*-edge from *P* to *A*, we get the same situation as above, although it is obtained as a combination of other modification operations.

Obviously, there are many other interesting conflicts that can be (and will be) detected using our formalism. For example, we will have similar conflicts with *«generalisation»*-edges between *«interface»*-nodes. However, we will not discuss all possible evolution conflicts in detail here.

VI 3.6 CONCLUSION

In the customisation to class collaborations of section VI . 2 , most of the domain-specific type constraints could be defined directly in the type graph. In the customisation to UML class diagrams this is no longer the case. Especially the constraints for dealing with the generalisation relationship turned out to be fairly complex, because they needed to take the transitive closure of *«generalisation»*-edges into account. This complexity was also reflected at the level of domain-specific applicability and evolution conflicts.

Although we have already identified a number of interesting evolution conflicts, further experiments are still needed to find more.

VI 3.7 FUTURE WORK

VI 3.7.1 Possible Extensions

Although both customisations we have encountered until now (collaborating classes and class diagrams) overlap in some way, they deal with orthogonal aspects of the design of a software system. With collaborations, the emphasis lies on the dynamic interaction between participants by means of operation invocations. Class diagrams only look at the static aspects, but additionally address issues like inheritance between classes and attributes of classes. Combining both approaches in one model would be very interesting, as it would allow us to model the subtle interaction between the inheritance mechanism and the message sending mechanism. This would make it possible to express super sends and self sends (with late binding), allowing to detect more sophisticated evolution conflicts.

Besides this, other interesting extensions can be conceived. Each of these extensions will have an impact on the possible contract types and corresponding conflicts.

- We could introduce *«aggregation»* (or composite association) as a subtype of *«association»*. This is necessary if we want to model part-whole relationships. This extension could lead to the introduction of new interesting domain-specific contract types. For example, [Johnson&Opdyke93] defines and motivates two behaviour-preserving transformations *AggregationToGeneralisation* and *GeneralisationToAggregation* that allow us to change an *«aggregation»*-relationship into a *«generalisation»*-relationship and vice versa. These transformations can be defined in terms of our primitive contract type *EdgeRetyping*. *«aggregation»* can also lead to new interesting evolution conflicts. For example, *EC5: Cycle introduction conflict* is probably a conflict if two *«aggregation»*-relationships are introduced independently in the opposite direction between two classes *C* and *D*, since it would mean that *C* is a part of *D*, while *D* must be a part of *C*.
- We can add *«implements»*-edges between classes and interfaces to specify which classes implement which interfaces. In UML, this can be achieved by means of a *Dependency* relationship. This leads to an additional constraint between classes and interfaces: a class that implements an interface must

understand at least all operations understood by the interface. As a result, changes in the interface can affect the classes that implement the interface and vice versa. For example, by removing an operation from a class, it is possible that the class no longer implements the interface. This will give rise to a number of new domain-specific applicability conflicts.

- We can add type information to attributes. To each attribute a type can be attached that corresponds to an existing class or interface in the diagram.
- We can take the object level into account, by specifying objects as instances of classes. Each object needs to specify values for all the attributes understood by the class of which it is an instance.
- We can attach method implementations to each operation in a class. We can also add additional information to the signature of an operation, like return types and arguments.
- We can deal with visibility and other issues by attaching a tag (public, private, protected, abstract, concrete, final, static, …) to each attribute and operation. Tags can also be associated to classes in an analogous way. [Cornelis97] investigated the effect of these tags on the possible evolution conflicts.
- We can introduce packages and their import/export mechanisms to reduce the inherent complexity of large class diagrams.
- We can try to cope with template classes (á la C++) by introducing some template mechanism.

VI 3.7.2 Other UML Diagrams

The approach that was taken here to customise the domain-independent reuse contract framework to UML class diagrams can easily be adopted to deal with other kinds of UML diagrams as well. Therefore, the following steps need to be taken:

- First, we need to identify those parts of the UML metamodel that are necessary for describing the chosen UML diagram.
- Next, the required metaclasses of the UML metamodel must be mapped to nodes and edges in a type graph, and well-formedness rules on these metaclasses (usually expressed in OCL) must be translated into constraints on this type graph.
- Then, we need to specify a partial order of node types and edge types based on the generalisation relationships in the UML metamodel.
- As a subsequent step, domain-specific contract types need to be defined that correspond to useful and typical ways of modifying or evolving parts of the considered UML diagram.
- Finally, the possible domain-specific applicability and evolution conflicts need to be investigated.

In [Mens&al99a] we have already investigated how the reuse contract approach can be used to add support for evolution to *UML interaction diagrams*. Because these diagrams can be seen as an extension of the customisation to class collaborations of section VI . 2 , we are convinced that our formal framework can be customised immediately to UML interaction diagrams. Other interesting customisations would be *UML use case diagrams* and *UML statechart diagrams*. An attempt to add support for evolution to use cases has been made in [Ecklund&al96], where *change cases* were proposed as descriptions of future requirements, and indicators of potential directions of future development. An early attempt to deal with support of evolution for statecharts has already been presented in [Mens&Steyaert97].

Because the UML metamodel itself is described using class diagram notation, we could also apply our reuse contract formalism to express evolution of UML itself. Indeed, although UML is an industry standard, it keeps on evolving. In late 1997, version 1.1 was published. The current version of UML is version 1.4. A problem with expressing evolution of the UML metamodel is that parts of it are still defined in natural language, while other parts are expressed in OCL. Both specification mechanisms still do not have a formal semantics [Gogolla&Richters98, Hamie&al98], so that it is impossible to detect potential evolution conflicts in a formal way. Fortunately, attempts to formalise the UML semantics are being made [Richters&Gogolla98, Schro&France97, Breu&al97].

VI . 4 SOFTWARE ARCHITECTURES

This section elaborates how the formal reuse contract framework can be customised to the domain of software architectures. The ideas presented here are based on ongoing research by Natalia Romero and Kim Mens. In her masters thesis, Natalia will investigate if and how the reuse contract model can be used to deal with evolution of software architectures. In his Ph. D. dissertation, Kim Mens will investigate the topic of automated reasoning about the impact of unanticipated evolution of software architectures on lower-level software artifacts and vice versa. Natalia and Kim's preliminary research results are included here as extra evidence that the formalism can be customised to many software engineering domains.

VI 4.1 INFORMAL DISCUSSION

When designing a software system it is very important to agree upon an adequate *software architecture*. Software architectures emerged as a natural evolution of design abstractions, as engineers searched for better ways to understand their software and new ways to build larger, more complex software systems [Shaw&Garlan96]. Software architectures provide a high level view on the overall structure and design of a software system. They describe a software system in terms of a collection of architectural elements – typically referred to as the *components* of the architecture – together with a description of the interactions and relationships among those elements – sometimes called the *connectors* – and a set of constraints on these components and connectors [Perry&Wolf92, Garlan&Shaw93, Garlan95, Schwanke&al96, Buschmann&al96]. Components represent the primary units of computation and data storage (state).

As a typical example, Figure 45 illustrates the architecture of a traditional sequential compiler [Aho&al86, Perry&Wolf92, Shaw&Garlan96]. Such a compiler typically distinguishes the following phases: lexical analysis, syntactic analysis (parsing), semantic analysis (such as type checking), optimization and code generation. Each phase takes data in some form as input, processes and transforms it, and passes the transformed data on to the next phase. The "components" in this architecture are the processing elements such as a lexer (lexical analyser), parser (syntactic analyser), semantor, optimizer and code generator. The lexical analyser takes source code text as input and generates a lexical token stream, which is transformed into a parse tree by the syntactic analyser, and then transformed into intermediate code and so on, until machine code is produced. This architecture is an example of a pipeline style architecture. Because the connectors in a pipeline architecture are very simplistic and merely transport data from one component to another, we represented them as simple lines in Figure 45.

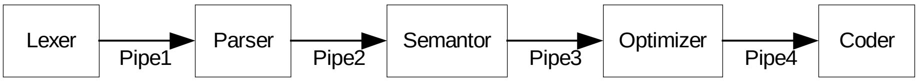


Figure 45: Sequential Compiler Architecture

The choice of an architecture should be carefully considered, since changes to it usually require complex and costly changes to substantial parts of the system. Nevertheless, architectural evolution is unavoidable, since constantly changing requirements sometimes force the software architecture to be revised [Poulin96, Bosch98]. As an example of such an evolution we will make a slight revision of the sequential compiler architecture. Indeed, this architecture is not completely accurate because most traditional compilers do not completely follow the pipeline style. Additionally, they use a separate symbol table that is created during lexical analysis and used or updated during subsequent phases [Shaw&Garlan96]. This gives rise to the architecture of Figure 46. In this revised architecture, two different kinds of components can be distinguished: the computational components that transform data into another form, and a memory component for storing the symbol table. Instead of one kind of connectors for representing data flow, now we also have two other kinds: a connector for creating the symbol table and several connectors for accessing and updating the symbol table. For reasons of simplicity, however, we have not made a visual distinction between these different kinds of components and connectors in Figure 46. Yet another evolution would be to add an error handler which interacts with all phases.

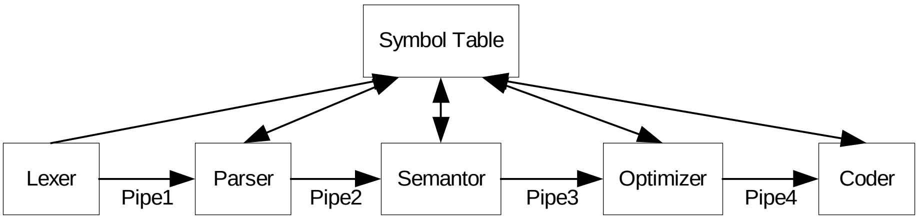


Figure 46: Revised compiler architecture

VI 4.2 TERMINOLOGY

The basic entities in an architecture are *components* and *connectors*. Connectors can be attached to the components using one or more *ports* that each define a logically separable point of interaction with the environment, and that can be regarded as the external interface of a component. Similar to the ports of components, connectors have an interface consisting of *roles*. Note that, in the examples of Figure 45 and Figure 46 we have purposefully omitted the explicit use of ports, connectors and roles, as it would only clutter the example. We will later see how these examples can be specified in more detail.

In current literature on software architectures, there is some discussion on the need for *both* components and connectors [Allen&Garlan97]. Is it necessary to have an architectural model which includes both a notion of components and connectors, or should we, for example, express connectors as a special kind of components? Both approaches have their own specific advantages, and examples of architectural description languages following either of these approaches can be given.

The advantage of *not* making a distinction between components and connectors is that it significantly simplifies the architectural language. Also, if there is only one kind of element, it is easier to define a semantical mapping for it. If the primary goal of the language is to support reasoning and formal manipulation, the language should strive for minimality, and preferrably have only one kind of architectural entity.

On the other hand, although from a formal point of view explicit connectors are not a strict necessity, in practice they are indispensable. If the primary goal of the architectural language is to provide a vehicle of expression that matches the intuitions of users, it is better to include both components and connectors. For example, the explicit treatment of software connectors makes it easy for an architect to separate communication issues from computation in a system.

Because the main concern of Natalia Romero and Kim Mens – who try to apply the formal framework described in this dissertation to the domain of software architectures – is the development of a formalism to support reasoning about evolution of software architectures, they take the more simplistic approach. In other words, in their architectural model, they do not make a distinction between components and connectors. There are only architectural *elements* which can be either *primitive* or *composite*. Primitive elements are pre-defined in the model, whereas composite elements are defined in terms of a subarchitecture. Elements have *gates*, just like components have ports and connectors have roles. Elements can be related to other elements by *linking* their gates together. When a composite element is defined in terms of an architecture of many interrelated more elementary elements, we also need to specify how the external interface of the composite element is connected to these internal elements. For this purpose, the gates of the composite element are *bound* to (some) gates of (some of the) internal elements.

VI 4.3 TYPE CONSTRAINTS

VI 4.3.1 Type partial order

Let us explain now how we can model architectures in terms of nodes and edges. Based on the above terminology, we can distinguish three types of nodes: «*architecture*»-nodes, «*element*»-nodes and «*gate*»-nodes. Every «*element*»-node is either a «*comp-el*»-node (composite) or a «*prim-el*»-node (primitive). The distinction between both is that a «*comp-el*»-node is required to have an «*architecture*»-node (which corresponds to its implementation) nested in it. There are two types of edges: «*link*»-edges that connect «*gate*»-nodes nested in «*element*»-nodes at the same level, and *«binding»*-edges that connect a «gate»-node nested in a *«comp-el»*-node to a *«gate»*-node nested in one of the internal elements of the composite element.

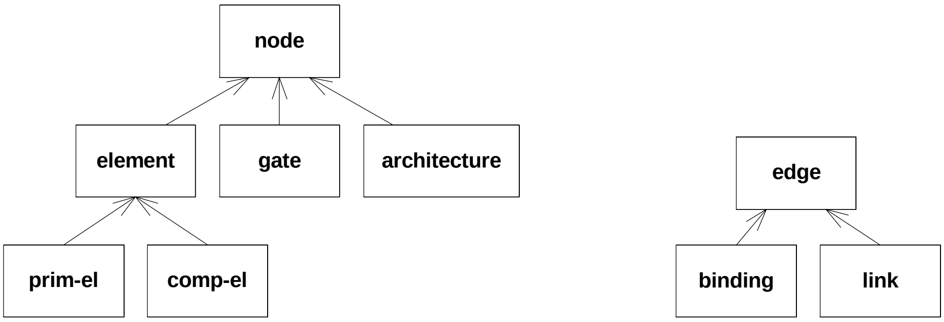


Figure 47: Type partial order for software architectures

The partial order of these node types and edge types is given in Figure 47. Using these types, the graph representation of (part of) the sequential compiler is given in Figure 48. This picture looks more complex because it explicitly represents gates, while these were omitted for the sake of simplicity in the previous examples. Another reason why the example looks more complex is because the *SequentialCompiler* architecture is nested inside a composite element called *Compiler*. This facilitates evolution afterwards, because we only need to modify the internal architecture and update the bindings, without this having a substantial influence on other elements that use this *Compiler*.

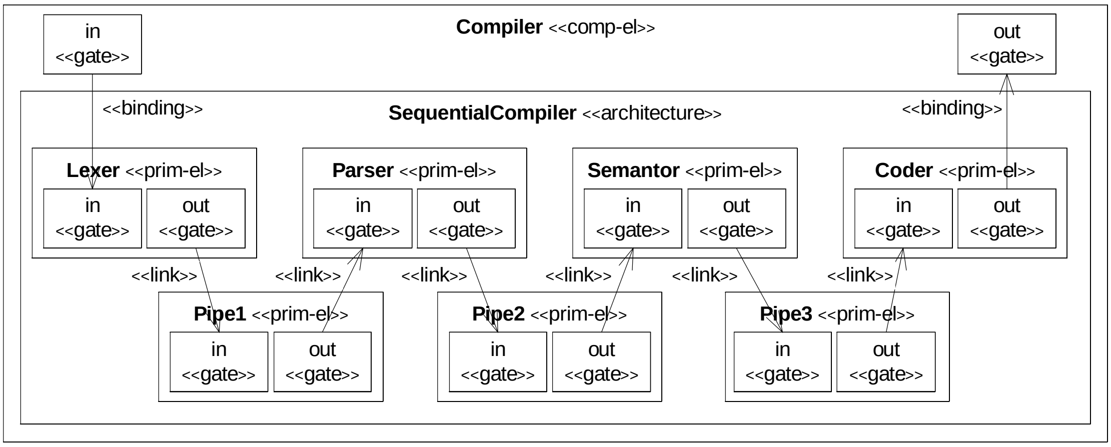


Figure 48: Graph representation of compiler architecture

Note that, although we have made no explicit distinction between components and connectors, the approach does not really disallow it. If necessary a distinction could be made by introducing two new subtypes of *«element»*, namely *«component»* and *«connector»*.[7] Similarly, we would need to define *«port»* and *«role»* as subtypes of *«gate»*. Of course, because *«prim-el»* and *«comp-el»* are also subtypes of *«element»* we would also need to introduce node types for primitive components, composite components, primitive connectors and composite connectors.

Another important remark regarding node types, is that sometimes it may be useful to include style-specific or even architecture-specific node types. We already encountered an example of this in the sequential compiler architecture which followed the pipeline style. In this architectural style we distinguish *«pipe»*-connectors and *«filter»*-components.

VI 4.3.2 Type graph

In Figure 49, the type graph is represented. *«gate»*-nodes must be nested inside *«element»*-nodes. *«comp-el»*-nodes should contain exactly one nested *«architecture»*-node (that implements the

[7] An alternative approach would be to represent components as nodes and connectors as edges, but this has the strong disadvantage that it is impossible to define composite connectors, or connectors with arity more than two such as broadcast connectors or busses.

composite element). *«architecture»*-nodes can also occur at top level. *«element»*-nodes must be nested inside an *«architecture»*-node. *«link»*-edges and *«binding»*-edges are only allowed between gates. Because we only allow one *«link»*-edge or *«binding»*-edge between any two *«gate»*-nodes, we will always attach the empty label ε to these edges.

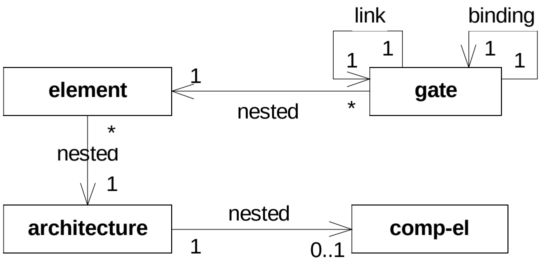


Figure 49: Software architecture type graph

Extra constraints will be needed at the moment we introduce *«component»*-nodes, *«connector»*-nodes, *«port»*-nodes and *«role»*-nodes. For example, *«port»*-nodes must be nested in *«component»*-nodes, *«role»*-nodes must be nested in *«connector»*-nodes, *«link»*-edges can only connect *«port»*-nodes to *«role»*-nodes or vice versa, and *«binding»*-edges can only connect *«port»*-nodes with *«port»*-nodes or *«role»*-nodes with *«role»*-nodes.

Apart from the above constraints there may also be more specific type constraints when we consider a specific architectural style such as the pipe-and-filter style. For example, every port of a filter must be either an in-port or an out-port and similarly for roles of pipes. Furthermore, the pipeline style, which is a further specialisation of the pipe-and-filter style, imposes even more constraints. Here, the used components are *stages* instead of filters. The distinction with filters is that stages can only have one input and one output port.

VI 4.3.3 Additional type constraints

There are some additional nesting constraints that cannot be expressed immediately in the type graph because they deal with more than two nodes and edges at the same time. The two constraints mentioned below restrict the use of *«link»*-edges and *«binding»*-edges. In order to understand the constraints better, compare them with how the edges are used in the concrete example of Figure 48.

Constraint C_1: *«link»*-edge

> *«link»*-edges can only connect *«gate»*-nodes that are nested in *«element»*-nodes that are both nested inside the same *«architecture»*-node

Constraint C_2: *«binding»*-edge

> *«binding»*-edges can only connect a *«gate»*-node nested in a *«comp-el»*-node with a *«gate»*-node nested in an *«element»*-node that is nested inside an *«architecture»*-node which is nested in the *«comp-el»*-node.

VI 4.4 Domain-specific contract types

VI 4.4.1 Primitive Contract Types

According to [Oreizy96], the four fundamental operations for changing a software architecture are addition and removal of components and connectors. Although their names may vary, those operators are provided by most reconfiguration languages. The domain-specific contract types defined by Natalia and Kim for modifying software architectures are: adding, removing and renaming architectures, elements and gates; adding or removing links and bindings; and refining or coarsening an element. All these operations can be expressed easily in terms of the domain-independent primitive contract types.

AddArchitecture(A)

> **Description.** Add an empty architecture *A* at top level.
> **Corresponds to:** *Extension(A, «architecture»)*

RemoveArchitecture(A)

Description. Remove an empty architecture *A* from top level.
Corresponds to: *Cancellation(A, «architecture»)*
Remark. This operation could be extended to remove nonempty architectures as well, as long as they have no bindings to external elements.

AddElement(A.E)

Description. Add an element *E* to an architecture *A*.
Corresponds to: *Extension(A.E, «prim-el»)*
Remark. This operation can only be used for adding primitive elements to an architecture. They can be made composite afterwards by performing the operation *ConcretiseElement*. Because of the *nested* edge in the type graph, *A* must be of type *«architecture»*.

RemoveElement(A.E)

Description. Remove an element *E* from an architecture *A*.
Corresponds to: *Cancellation(A.E, «element»)*

In all following contract types, we still require elements to be nested in an architecture, but we will no longer qualify the elements by the label of the architecture in which they are nested to make the examples more readable.

AddGate(E.G)

Description. Add a gate *G* to an element *E*.
Corresponds to: *Extension(E.G, «gate»)*
Remark. Because of the *nested* edge in the type graph, *E* must be of type *«element»* (or a subtype thereof).

RemoveGate(E.G)

Description. Remove a gate *G* from an element *E*.
Corresponds to: *Cancellation(E.G, «gate»)*
Remark. The gate can only be removed if it is not linked to any other gate.

ConcretiseElement(E, A)

Description. Introduce a new empty architecture *A* as the implementation of a primitive element *E*.
Corresponds to: *[Extension(E.A, «architecture»), RetypeNode(E,«prim-el»,«comp-el»)]*
Remark.
The architecture *A* that is being nested in the element *E* should be empty, but can be filled in afterwards, for example by using the composite contract type *CopyArchitecture* that will be introduced later.
As a side-effect of refining a primitive element, its type must be changed from *«prim-el»* to *«comp-el»*, since only composite elements are allowed to contain architectures.

AbstractElement(E, A)

Description. Remove an empty architecture *A* as the implementation of a composite element *E*.
Corresponds to: *[Cancellation(E.A,«architecture»), RetypeNode(E,«comp-el»,«prim-el»)]*
Remark. Once the architecture is removed, the composite element becomes primitive again.

AddLink(E_1.G_1, E_2.G_2)

Description. Introduces a new link from gate G_1 in element E_1 to gate G_2 in element E_2.
Corresponds to: *Refinement(ε, E_1.G_1, E_2.G_2, «link»)*

RemoveLink(E_1.G_1, E_2.G_2)

Description. Removes an existing link between the gates G_1 and G_2.
Corresponds to: *Coarsening(ε, E_1.G_1, E_2.G_2, «link»)*

In a similar way, there are operations ***AddBinding(E_1.G_1, E_2.G_2)*** and ***RemoveBinding(E_1.G_1, E_2.G_2)*** for adding and removing a binding between gates G_1 and G_2.

RenameElement(E_1,E_2)

Description. Renames an element with name E_1 to E_2.
Corresponds to: *RelabelNode(E_1, «element», E_2)*

In a similar way, there are operations ***RenameGate(G_1, G_2)*** and ***RenameArchitecture(A_1, A_2)*** for renaming a gate or an architecture. Renaming is unnecessary for bindings or links, since these always have an empty label ε.

VI 4.4.2 Composite Contract Types

There is one domain-specific modification operation which is substantially more complex than the other ones: copying a particular architecture to a different one. In order to deal with this operation, a new composite contract type needs to be defined.

CopyArchitecture(A_1, A_2)

Description. Copies the contents of an architecture A_1 to a different one which is empty at first. This operation is achieved by recursively copying all nodes which are nested in A_1 to A_2, and also copying all edges between these nested nodes. Basically, this can be expressed in terms of a composite extension and a composite refinement.

A related composite contract type ***MoveArchitecture(A_1, A_2)*** can be defined that not only copies the architecture, but at the same time deletes the first occurrence of the architecture. This corresponds to moving the architecture from one place to another.

Because these operations are useful in other domains as well, it would be a good idea to introduce them as new domain-independent composite contract types ***CopyContents*** and ***MoveContents***.

Other useful domain-specific composite contract types can be defined. An example is the replacement of a path of communication between two elements by a different one. In the case of the architectural definition of the sequential compiler, for example, one might decide that the *Optimiser* is not necessary, and should be bypassed. Of course, the *Semantor* and the *Coder* should still be connected to each other. A domain-specific composite contract type *Reconnect(Semantor.out,Coder.in,Pipe,in,out)* can be used for this purpose. It directly connects *Semantor* to *Coder* via a new intermediate element *Pipe*, instead of needing to go through the *Optimizer*. The exact definition of *Reconnect* is given below:

Reconnect(S.G_1 T.G_2, C, G_3, G_4)

Description. Redirect the target of a link from a gate *S.G_1* in source element *S* to a new gate *C.G_3* in an intermediate element *C*, and replace the source of a link from a gate *T.G_2* in target element *T* to a new gate *C.G_4* in intermediate element *C*.
Corresponds to:
[RedirectTarget(ε,S.G_1,C.G_3,«link»), RedirectSource(ε,T.G_2,C.G_4,«link»)]
Remark.
Before reconnecting, the gates G_3 and G_4 should not be linked to any other gates.
In the above definition, only 4 arguments are provided to *RedirectTarget* and *RedirectSource* instead of 5, as required in the definition of section V 3.2.1 on page 130. This is for convenience, because the fifth argument can be automatically calculated since there can be only one link starting from (or arriving in) each gate.

The above operation does not actually delete the elements that are removed from the path from *S* to *T*, because these elements can still be referred from other parts of the architecture, and checking whether this happens or not is only possible by attaching additional (complicated) constraints to the composite contract type.

VI 4.5 Domain-specific conflicts

Many of the conflicts that are detected by the domain-independent formalism are too simple to be practical in the customisation to software architectures. We are more interested in conflicts between elements than in low-level conflicts that appear at the level of links between the gates that are nested in those elements. Therefore, the basic evolution conflicts need to be translated to a more abstract level. This can be achieved by defining *derived edges* as combinations of more primitive ones, as introduced in section V 5.2. Conflicts can then be detected at the level of these derived edges.

In fact, all the edges mentioned in Figure 45 can be derived from the low-level edges between gates in Figure 48. The technique is illustrated in Figure 50. First, *«link»*-edges between *«gate»*-nodes can be promoted to edges between the *«element»*-nodes in which they are nested (Definition 55). Next, a sequence of two such *«promoted»*-edges can be replaced by a *«transitive»*-edge (Definition 57). In other words, at a higher level, pipes are no longer represented as nodes, but as derived edges.

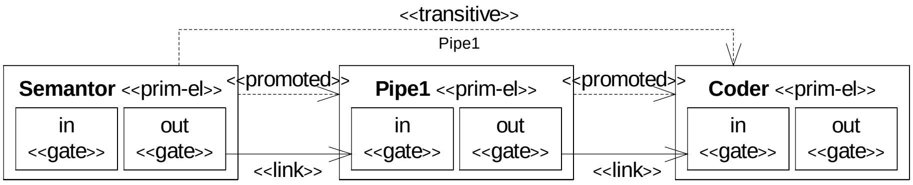


Figure 50: Derived edges between architectural elements

In the above example, an evolution conflict between derived edges would arise if independent evolvers make the following interacting modifications. The first evolver decides to replace the *Coder* element by a new element *Coder2*. At a high level, this corresponds to the following contract type:

RedirectTarget(Pipe1,Semantor,Coder,«transitive»,Coder2)

The second evolver independently decides to put an *Optimizer* between the *Semantor* and the *Coder*. This is achieved by performing the following composite contract type:

AddElement(Optimizer), AddGate(Optimizer.in), AddGate(Optimizer.out),
RedirectSource(Pipe1,Semantor,Coder,«transitive»,Optimizer),
Refinement(Pipe2,Semantor,Optimizer,«transitive»)

When both modifications are merged they lead to an evolution conflict because of an undesired interaction between *RedirectTarget* and *RedirectSource*. More specifically, both composite contract types include a *Coarsening(Pipe1,Semantor,Coder,«transitive»)*, and thus give rise to an AC_6: Double coarsening conflict. Note, however, that this is only an applicability conflict at the level of derived edges. At the level of links between gates, no conflict will be detected. This allows us to conclude that, although our basic conflict detection mechanisms can still be used, they sometimes need to be applied at a higher level in order to find useful conflicts.

In the same way as explained above, many other interesting conflicts can be detected.

VI 4.6 Conclusions

Although further research and experiments are still needed, especially for the conflict detection part, we have illustrated that our formal framework can be customised to the domain of software architectures as well. However, during the customisation, a number of shortcomings of the current formalism have been identified.

The domain-specific customisation to software architectures motivated the need of a new composite contract type *CopyContents* and *MoveContents* (which copy or move the contents of a node recursively to a new node) that should be defined in the domain-independent formalism, as it is useful in other domains as well.

Another problem we encountered was that the evolution conflicts that are detected by the domain-independent formalism are too low level. In order to find practically interesting conflicts, we need to scale up the conflicts by making use of *derived edges*. While we can still apply the same conflict detection techniques, it enables us to find useful conflicts that cannot be detected at the lower level. We are currently experimenting with a customisation of our domain-independent PROLOG implementation to the domain of software architectures in order to find more useful evolution conflicts in this domain.

Although we have been able to express the concepts in a software architecture in terms of node types and edge types in a type graph, we did not deal with the notion of *architectural styles*. Every architectural style (such as the pipe-and-filter style) will have its own specific concepts that can be defined in terms of the general architectural concepts. Therefore, we need to extend the general type graph for architectures with more specific types and constraints needed for the architectural style under consideration. In other words, we need to be able to customise the domain-specific type graph to represent different architectural styles. One possibility would be to extend the reuse contract formalism to type graphs as well, thus allowing a type graph to evolve with the architecture (and to have conflict checking at that level also). Using this approach, every architectural style would have its own specific type graph, which is an extension of the original type graph representing the domain of software architectures. Of course, for each architectural style, new modification operations, constraints and conflicts need to be specified in exactly the same way as we did for the domain of software architectures itself.

VI 4.7 Related Work

Current apporaches to architectural evolution tend to focus mainly on how to deal with *run-time* evolution of software architectures [Kramer&Magee98, Oreizy&Taylor98, Wermelinger98]. With run-time evolution, also called *dynamic* evolution, the architecture is dynamically modified while the software is running, without compromising application integrity. Because of these strong requirements, run-time approaches restrict themselves largely to *anticipated* evolution. The architecture is only allowed to evolve in ways that can be foreseen in advance. Reuse contracts take a completely opposite approach, by allowing all kinds of evolution, and detecting potential problems afterwards.

Many approaches that formally describe software architectures explicitly use labelled graphs and graph rewriting to represent architectural elements and their interconnections [LeMétayer98, Wermelinger98, Hirsch&al99, Wermelinger99]. For example, in [Wermelinger99], a reconfiguration rule is defined as a labelled graph production $P: L \rightarrow R$, where the labelled graphs L and R represent software architectures. A reconfiguration step $G \Rightarrow_P H$ is a direct derivation from a given architecture G to a new architecture H.

Since we considered evolution of UML diagrams in section VI . 3 , we will also briefly discuss the relation between UML and software architectures. There is almost no support to deal with software architectures in UML. Several proposals have recently been made to cope with this problem. For example, the ROOM language incorporates concepts that constitute a domain-specific architecture description language, and a proposal to incorporate it in the UML has been given in [Selic&Rumbaugh98]. As another example, [Robbins&al98] shows how substantial elements of architectural models can be incorporated in the UML. Many other attempts for trying to describe software architectures using UML have been made [Hofmeister&al99, Medvidovic&Rosenblum99, Fradet&al99].

VI . 5 Summary and Future Work

VI 5.1 Summary

This chapter illustrated that our formal framework for software evolution is domain-independent. To this aim, it was applied to class collaborations and UML class diagrams. We have also motivated how the formalism could be customised to other kinds of UML diagrams, as well as to software architectures. In this way we have illustrated that reuse contract techniques for dealing with software evolution, more specifically the detection of potential evolution conflicts, are independent of a specific domain. This gives us evidence that the formal reuse contract framework can be customised to all phases of the software life-cycle, ranging from requirements analysis to implementation. Obviously, more experiments are needed to further validate this claim.

Let us now briefly recapitulate which actions need to be undertaken in order to customise the domain-independent reuse contract formalism to a specific domain:

- Identify the different *node types* and *edge types* required to deal with domain-specific concepts.
- Define a *partial order* on the node types and edge types.
- Specify the *domain-specific type graph* and any other additional type constraints.
- Express *domain-specific primitive contract types* in terms of the domain-independent ones.
- Define *domain-specific composite contract types*, in terms of domain-specific as well as domain-independent primitive contract types.
- Customise the *applicability conflicts*. Give new names to the domain-independent applicability conflicts so that they are more meaningful in the specific domain, and define new *domain-specific applicability conflicts* that arise if domain-specific type constraints are invalidated when performing an evolution step.
- Customise the *evolution conflicts*. Specify which of the domain-independent evolution conflicts are relevant in the domain, and which ones can be ignored. Also give an intuitive name to each of the relevant evolution conflicts.

The effect of customisation on the possible applicability and evolution conflicts can be summarised as follows:

- Because of our very general definition and characterisation of evolution conflicts, *no new domain-specific evolution conflicts will be introduced*. All evolution conflicts can be detected using our basic conflict detection mechanism, although our domain-independent evolution conflicts could be fine-tuned further to make a distinction between *Refinements* and *Coarsenings* in particular cases. Also, sometimes *derived edges* are needed to detect more meaningful high-level evolution conflicts. This was the case with the customisation to software architectures, where the detected evolution conflicts were too primitive to be practical. In order to detect more meaningful conflicts at a higher level of abstraction, *derived edges* were defined in terms of the more primitive ones.
- *Domain-specific well-formedness constraints*, imposed by the type graph, *can give rise to new domain-specific applicability conflicts*. A contract type is not applicable if the result graph no longer satisfies the type constraints (while the initial graph does).
- *Some domain-independent evolution conflicts can coincide with domain-specific applicability conflicts*. In other words, by imposing particular domain-specific type constraints, it is possible that situations that gave rise to an evolution conflict in the domain-independent framework, now give rise to an applicability conflict in the domain-specific customisation.
- *Some domain-independent evolution conflicts can be ignored in the specific domain*, while others only need to be detected in particular cases, depending on the type of node or edge that is involved. This allows us to reduce the number of evolution conflicts that are detected by the domain-independent conflict detection algorithm to a more manageable number. Indeed, the formal conflict detection approach of chapter IV was based on a “worst case” scenario, and generated conflicts for every potentially dangerous situation. By resorting to domain-specific knowledge, many of these situations can be identified as safe, so that the corresponding conflicts do not have to be detected.

VI 5.2 SHORTCOMINGS

The results put forward in this chapter indicate that the formal framework proposed in this dissertation is sufficiently general to be customised to different domains. Nevertheless, some extensions are still required:

- In the customisation to class collaborations there was a need to subdivide the domain-independent evolution conflicts, depending on whether they were caused by a *Refinement*, *Coarsening* or *EdgeRetyping*.
- In the customisation to UML class diagrams, *hyperedges* were required to deal with n-ary associations.
- The conflicts that could be detected for UML class diagrams were more complex than the ones for class collaborations, because the inheritance mechanism required us to make use of the transitive closure of the *«generalisation»*-edges to express particular constraints. Because these constraints could not be expressed simply using the type graph, the need arose to deal with more complex constraints in a formal way, in order to allow us to detect more complex conflicts in an automatic way.
- A more important result that can be concluded from the customisation to software architectures is that *particular domain-specific customisations can be considered as frameworks themselves*. For example, the customisation to software architectures could be considered as a framework that can be customised to incorporate different architectural styles. This requires us to repeat the same techniques at a more concrete level. It also requires us to apply the reuse contract approach at the level of type graphs (instead of ordinary graphs), since customisation of a domain to some subdomain involves customisation of the type graph to deal with more specific node types, edge types and constraints.

When customising the framework to other domains, new limitations might appear. Therefore, further experiments are still needed, preferrably in domains that are significantly different from the ones we already considered.

VII . CONCLUSION

This chapter concludes the dissertation by summarising the contributions of the thesis, and indicating potential areas of future research.

VII . 1 MAIN CONTRIBUTIONS

VII 1.1 OBJECTIVES OF THE THESIS

In this dissertation we proposed labelled typed nested graphs and conditional graph rewriting as a formal foundation for reuse contracts, and showed how the formalism could be used to deal with evolution in a *domain-independent* and *scalable* way.

VII 1.1.1 Domain Independence

In order to illustrate the domain independence, we have customised the formal reuse contract framework to different problem domains: collaborating classes [Lucas97], UML class diagrams and software architectures. This illustrates that our formalism *provides a framework to reason about evolution in a domain-independent way,* which was the main objective of our thesis.

Let us briefly recapitulate which actions need to be taken in order to customise the formal framework to a specific domain:

- map the different kinds of elements in the domain to different *node types*
- map the different kinds of dependencies (or relationships) between elements in the domain to different *edge types*
- define a partial order on these node types and edge types to specify the subtyping relationships
- define a domain-specific *type graph* and additional *type constraints* to specify extra well-formedness rules on the domain-specific graphs
- give domain-specific names to the relevant domain-independent primitive contract types and predefined composite contract types
- if necessary, define new domain-specific primitive and composite contract types
- give domain-specific names to the domain-independent applicability conflicts
- identify new domain-specific applicability conflicts that are induced by the domain-specific well-formedness rules imposed by the type graph and type constraints
- specify which domain-independent evolution conflicts should be ignored, e.g., based on the type of the edges and nodes that are involved in the conflict. Some evolution conflicts due to primitive contract types may also be ignored if these primitive contract types are part of a larger composite contract type.
- give domain-specific names to the remaining domain-independent evolution conflicts
- if necessary, identify new domain-specific evolution conflicts
- specify domain-specific conflict resolution rules

VII 1.1.2 Scalability

A second objective of our dissertation was to make the reuse contract formalism *scalable*. This was achieved in different ways:

- *Composite contract types* were defined as a sequential composition of other contract types. A *normalisation algorithm* was introduced to remove redundancy in arbitrary evolution sequences. Both techniques allowed us to reduce the total number of evolution conflicts in particular situations.
- A *transitive closure mechanism* was used to deal with indirect dependencies and their corresponding conflicts. This allowed us to detect more complicated kinds of conflicts.
- A *nesting mechanism* was used to deal with the inevitable complexity of software artifacts. It gave rise to new contract types and new evolution conflicts.
- *Derived edges* were introduced to deal with evolution conflicts at higher levels of abstraction.

- By modelling reuse contracts as edges in a graph, it became possible to document reuse and evolution, which allowed us to analyse the impact of a particular evolution, and to deal with propagation of changes.

Besides these main objectives, the dissertation also contributed in many other ways to the research on software evolution, and reuse contracts in particular, in a theoretical as well as a practical way.

VII 1.2 THEORETICAL CONTRIBUTIONS

The formalism of conditional graph rewriting has shown to be an elegant formalism for expressing evolution of software artifacts. Each of the primitive reuse contract types was described as an elementary graph production. For many results about evolution we could rely on existing properties such as parallel and sequential independence (for dealing with applicability and evolution conflicts) and sequential composition of productions (for dealing with composite contract types). Most of these properties relied on the category-theoretical notion of pushouts.

An important improvement on previous approaches is that the proposed formalism allowed us to express *negative information* in the contract clauses of a reuse contract. If suffices to make use of *negative application conditions*, not only as preconditions or postconditions on graph productions, but also as invariants on the graphs to which these productions are applied. Applicability of the production then requires both the invariants and the preconditions to be satisfied. We have not fully exploited the possibility to deal with *negative graph invariants*, so this needs to be investigated further.

The primitive contract types were defined in an orthogonal way, which allowed us to specify *inverse contract types*, i.e., contract types where the preconditions and postconditions are each others inverse. This enabled us to define redundant and absorbing pairs of contract types, which were useful for defining normalisation of composite contract types. The *normalisation property* for composite contract types showed that any sequence of primitive contract types can be reduced to a minimal canonical sequence of primitive contract types. This normalised sequence was minimal in the sense that all redundant information was removed, making the evolution sequence more comprehensible. Moreover, because the normalised sequence often is significantly smaller, evolution conflicts can be detected more efficiently.

We made an *explicit distinction between syntactic and semantic inconsistencies* when combining two independently evolved versions of the same software artifact. Syntactic inconsistencies, corresponding to applicability conflicts, were defined in terms of *parallel independence*. Because we only dealt with a finite and well-defined set of primitive contract types, we were able to give a *complete characterisation* of all possible kinds of applicability conflicts, which were detected by breaches of applicability preconditions. Semantic inconsistencies, corresponding to evolution conflicts, were defined in terms of the notion of *pullback*. Again, a complete characterisation of possible evolution conflicts was given, which could alternatively be detected by identifying particular graph patterns in the resulting graph obtained when serialising both contract types.

The above approach for detecting potential evolution conflicts assumed a *"worst case" scenario*: for every situation that is potentially harmful, a conflict is generated. This results in a large number of detected evolution conflicts. When customising the formal framework to a specific domain, however, domain knowledge can be used to reduce the conflicts to a more manageable number.

VII 1.3 PRACTICAL CONTRIBUTIONS

While Carine Lucas established the basic ideas, motivation and terminology for reuse contacts in her Ph. D. dissertation [Lucas97], Koen De Hondt also addressed some more practical issues by proposing several tools to automate evolution support in an integrated software development environment [DeHondt98]. In order to apply reuse contracts to large software systems, tool support is an absolute necessity. For example, manually checking evolution conflicts is practically infeasible due to the size of software systems and the large number of evolution conflicts that can be detected. An advantage of reuse contracts is that they can be incorporated in tools in a fairly straightforward way. Because they are primarily based on static information, they do not have to rely on sophisticated techniques such as data flow analysis and deadlock detection to find interesting inconsistencies. On the other hand, reuse contracts can only detect *potential* problems because of these restrictions.

Our dissertation contributes to the practical goal of tool support by presenting several algorithms based on the formalism we proposed. Most of these algorithms have been incorporated in a prototype implementation of the reuse contract formalism developed in PROLOG. Moreover, this domain-independent PROLOG framework has been customised to the domains of class diagrams and software architectures to perform our experiments.

First a basic *conflict detection* algorithm was proposed to check applicability (or syntactic) conflicts as well as evolution (or semantic) conflicts. Obviously, detecting the latter kind of conflict is much more important. An extension of the conflict detection algorithm was proposed to deal with sequences of primitive evolution steps.

In order to make the evolution process more understandable, and to make the conflict detection algorithm more efficient, a *normalisation algorithm* was proposed. This algorithm allows us to remove redundant information in an arbitrary evolution sequence, thus making the evolution process easier to understand, and speeding up conflict detection.

Finally, an *extraction algorithm* can be used to extract reuse contract information if an arbitrary software artifact and its evolved version are given. This issue is not new, since it has also been addressed in [DeHondt98] in the context of reverse engineering.

VII . 2 FUTURE WORK

VII 2.1 ALGORITHMS AND TOOL SUPPORT

The conflict detection algorithm presented in this dissertation has an important practical shortcoming. Sometimes so many evolution conflicts will be detected that it becomes unfeasible to find out which of them are the more important ones that should be dealt with first. Although some conflicts can be ignored in a domain-specific customisation, the reduced number of domain-specific conflicts might still be too large to be manageable in practice. Therefore, techniques and heuristics should be developed that allow us to sort the evolution conflicts in order of importance. It is also necessary to find out which evolution conflicts become resolved automatically if a more important conflict gets resolved first. A typical example are higher-order conflicts that occur because of a first-order conflict somewhere else, due to the fact that the changes are propagated.

Although we did not formally support it, we briefly mentioned the need for a *conflict resolution algorithm*. It remains to be seen to which extent support for conflict resolution can be provided in a domain-independent way.

While we have presented the basic idea behind a normalisation algorithm, some more work needs to be performed before it can be applied in practice. For example, we still need to investigate how the algorithm behaves in the presence of type constraints, and how it can cope with composite contract types.

Obviously, all the above mentioned algorithms will never be used as a stand-alone tool. Instead, they should be incorporated in a CASE-tool or an integrated software development environment. This is necessary, since most of the current environments for object-oriented software development provide no or poor support for software evolution. Some small but promising experiments with CASE tools for UML indicate that integration of reuse contract tools in such an environment is feasible.

It is also very useful to integrate the reuse contract approach in a software configuration management system or, more specifically, a version management system. In this way semi-automated support can be provided when merging different versions of the same software artifact. Although several commercially available merge tools exist, none of them allow us to detect behavioural inconsistencies, while reuse contracts do.

As a final remark, it has never been the intention of this dissertation to address efficiency issues of the presented algorithms. We are aware of the fact that some of the algorithms are inefficient, and can be improved in many ways. Finding techniques and heuristics to optimise the algorithms is another important area of future work.

VII 2.2 ENHANCING THE EXPRESSIVENESS OF REUSE CONTRACTS

The expressiveness of the formal reuse contract framework developed in this dissertation can still be enhanced in many ways.

In chapter IV we briefly proposed two alternatives to the conflict detection approach that was taken in this dissertation: *making implicit evolution assumptions explicit*, and *making use of evolution invariants*. Each of these approaches can be defined easily in terms of conditional graph rewriting. It is worthwhile to work out these alternatives in more detail, to see if they can provide us new insights and results. Because the alternative approaches are complementary to the reuse contract approach, both techniques could be combined to obtain a more powerful conflict detection mechanism.

Next to positive (=required) and negative (=forbidden) constraints, it would be useful to deal with *optional information* as well. This will allow us to fine-tune the evolution conflicts, and allow us to determine in some cases that potential evolution conflicts are actual evolution conflicts. In [Schürr95], the use of optional nodes in a graph is discussed.

Another interesting extension would be to use *propositional application conditions* [Heckel95]. Instead of only allowing to express negative and positive application conditions, propositional conditions also include negations, conjunctions and disjunctions. This makes them more expressive, and thus more easy to use in practice.

In order to detect more sophisticated evolution conflicts it is also necessary to deal with complex constraints in a more formal way. This was clearly seen in the customisation to UML class diagrams, where the constraints involving *«generalisation»*-edges could only be defined in terms of a transitive closure. Because of these complex constraints, the associated evolution conflicts also became more complex. In [Fradet&al99], a poweful yet simple constraint language was introduced, which could be used to express more powerful constraints in a formal way, and thus allowing to detect more complex evolution conflicts automatically.

Another extension which is necessary but has not yet been considered is the ability to have *more generic reuse contracts*. This can be formally dealt with by making use of *generic graphs* (or *graph templates*), where the nodes and edges can be parameterised. This will clearly enhance the expressiveness of the formalism, but will also make the proofs more complex. For example, [Schürr95] reports on unsolved type checking problems that arise when dealing with parameterised edges.

A final extension would be to deal with graphs in a more behavioural way. If we use graphs as finite state machines, the visual equivalent of regular expressions, it might be possible to detect even more interesting evolution conflicts.

The disadvantage of each of the extensions proposed above is that they will all make the formal proofs more complex. The conflict detection algorithm will also become more sophisticated, and hence more difficult to integrate in tools.

VII 2.3 Further Validation of the Framework

In this dissertation we illustrated how the proposed formalism could be customised to deal with evolution in different problem domains: collaborating classes, UML class diagrams and software architectures. Nevertheless, further customisations to different domains are needed. Some interesting possibilities are mentioned below.

VII 2.3.1 Other UML Diagrams

In the specific customisation of the framework to UML class diagrams, we exploited the *UML metamodel* for providing us with a mapping of the essential model elements to node types and edge types in a graph. Currently, we are experimenting with this customisation of UML class diagrams in a large case study (about 600 classes) performed in collaboration with an industry partner.

In the same way as we have customised the formalism to UML class diagrams, we can provide support for evolution of other kinds of UML diagrams as well. Among the possible choices we have reuse contracts for use case diagrams and reuse contracts for statecharts. An early attempt to deal with the latter was already reported in [Mens&Steyaert97]. Customisation of the formal framework to these new domains might necessitate addition of new features to the formalism.

Note that, if we want to customise our formalism to different kinds of UML diagrams, it would be useful to follow the same approach as was taken with the customisation to software architectures and architectural styles. First, we can provide a customisation of the formal reuse contract framework to deal with general concepts in UML (such as *ModelElements*, *Relationships*, *Classifiers* and *Dependencies* which represent abstract metaclasses in the UML metamodel). Next, we can further customise this model to different subdomains, each representing one of the possible UML diagrams.

VII 2.3.2 Reuse Contracts for Type Graphs

This dissertation only discussed evolution of graphs, which are used to express arbitrary software artifacts. Sometimes, however, *evolution of type graphs* is also necessary. This is for example the case if we want to further customise the domain-specific customisation of software architectures to deal with different kinds of architectural styles. Each of the architectural styles requires new kinds of node types and edge types, and imposes additional constraints on the type graph.

In some sense, this can be considered as a kind of *meta-evolution*, since a type graph is a metagraph that expresses information about graphs themselves. Since the notions of graphs and type graphs are almost the same from a formal point of view, applying the reuse contract formalism to type graphs will probably not pose many technical difficulties. However, evolution of type graphs will have an impact on the domain-specific applicability and evolution conflicts that are detected at the level of graphs. Certain kinds of modifications that used to work fine can suddenly lead to a conflict because the type graph has evolved. To be able to deal with this, we also need to formally describe how evolution of type graphs may impact the possible evolution conflicts.

VII 2.3.3 Reuse Contracts For Co-Evolution

A challenging step is to see if the formalism can also be used to deal with *co-evolution.* This is necessary for reuse contracts to become an essential and integrated part of an evolutionary software development methodology. When a software artifact in a particular phase of the software life-cycle (e.g., design) evolves, all the corresponding artifacts in other phases (e.g., analysis and implementation) should be modified as well. In other words, related software artifacts in different phases should be kept "in sync". To be able to do this, the link between the different phases should be documented with reuse contracts, and conflicts should be detected when a software artifact in a particular phase evolves. This problem is closely related to the issues of traceability, compliance checking, and impact assessment.

VII 2.3.4 Reuse contracts for Non-OO Paradigms

An interesting topic of future research is to find out how well the reuse contract formalism can be applied to non OO-languages and methodologies. This should not pose many problems, since the essence of the formalism is that everything can be described by means of (nested) nodes and edges. Nowhere in our formalism did we rely on specific features of the object-oriented paradigm. Therefore, we are convinced that the ideas in this dissertation are also applicable to non OO systems.

VII 2.4 OTHER EVOLUTION APPROACHES

In this dissertation we have only illustrated that the reuse contract approach can be defined in a domain-independent way. The same technique could also be applied to many other approaches that deal with evolution in a specific domain, by translating their underlying ideas to a domain-independent formalism. Preferrably, the chosen underlying formalism should be the same as the one that we have decided upon here, so that the ideas from the different approaches towards evolution can be unified in one powerful domain-independent framework.

One approach which is particularly interesting is [Wiels&Easterbrook98], where a category-theoretical approach is taken to manage evolving requirement specifications. The proposed formalism allows to reason about the impacts of change on interconnected components, and also supports compositional (or incremental) verification. Moreover, because the ideas are expressed in category theory, there is good hope that the approach can be used for other domains as well.

VII . 3 BIBLIOGRAPHY

[Aho&al86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Allen&Garlan97] Robert Allen and David Garlan: *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, July 1997.

[Banerjee&al87] Jay Banerjee, Won Kim, Hyoung-Joo Kim and Henry F. Korth: *Semantics and Implementation of Schema Evolution in Object-oriented Databases.* SIGMOD RECORD, 16(3): 311-322, ACM Press, December 1987.

[Barbedette91] Gilles Barbedette: *Schema Modifications in the LISPO$_2$ Persistent Object-Oriented Language*, Proceedings of ECOOP '91, pp. 77-96, ed. Pierre America, LNCS 512, Springer-Verlag, 1991.

[Barbier&al98] Franck Barbier, Henri Briand, Bénédicte Dano and Stéphane Rideau: *The Executability of Object-Oriented Finite State Machines*. Journal of Object-Oriented Programming, 11(4): 16-24, SIGS Publications, 1998.

[Bassett97a] Paul G. Bassett: *Framing Software Reuse: Lessons From the Real World*. Yourdon Press Computing Series, ISBN 0-13-327859-X, Prentice Hall, 1997.

[Bassett97b] Paul G. Bassett: *Breaking Down The Complexity*. Object Magazine, September 1997.

[Bergstein94] Paul L. Bergstein: *Managing the Evolution of Object-Oriented Systems*. Ph. D. Dissertation, College of Computer Science, Northeastern University, 1994.

[Berzins94] Valdis Berzins: *Software Merge: Semantics of Combining Changes to Programs.* ACM Transactions on Programming Languages and Systems, 16(6): 1875-1903, ACM Press, November 1994.

[Binkley&al95] David Binkley, Susan Horwitz and Thomas Reps: *Program Integration for Languages with Procedure Calls*. ACM Transactions of Software Engineering Methodology 4(1): 3-35, ACM Press, January 1995.

[Bohner96] Shawn A. Bohner: *Software Change Impact Analysis for Design Evolution*. In [Bohner&Arnold96a], pp. 67-81.

[Bohner&Arnold96a] Shawn A. Bohner and Robert S. Arnold: *Software Change Impact Analysis*. IEEE Press, 1996.

[Bohner&Arnold96b] Shawn A. Bohner and Robert S. Arnold: *An Introduction to Software Change Impact Analysis*. In [Bohner&Arnold96a], pp. 1-26.

[Booch94] Grady Booch: *Object-Oriented Analysis and Design with Applications*. Second edition, Benjamin/Cummings, 1994.

[Bosch98] J. Bosch: *Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study*. Technical Report, 1998.

[Breu&al97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe and V. Thurner: *Towards a Formalization of the Unified Modeling Language*. Proceedings of 11th European Conference on Object-Oriented Programming, LNCS 1241, Springer-Verlag, 1997.

[Buffenbarger95] J. Buffenbarger: *Syntactic Software Merging*. Software Configuration Management: Selected Papers SCM-4 and SCM-5, J. Estublier (ed.), LNCS 1005, pp. 153-172, Springer-Verlag, 1995.

[Buschmann&al96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal: *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons, 1996.

[Butler97] Greg Butler: *A Reuse Case Perspective on Documenting Frameworks.* Technical Report, 1997.

[Canning&al89] Peter S. Canning, William R. Cook, Walter L. Hill and Walter G. Olthoff: *Interfaces for strongly-typed object-oriented programming*. Proceedings of OOPSLA ’89, editor N. Meyrowitz, ACM SIGPLAN Notices, 24(10): 457-467, ACM Press, 1989.

[Carrington&al90] D. A. Carrington, D. Duke, R. Duke, P. King, G. A. Rose and G. Smith: *Object-Z – An object-oriented extension to Z*. In Formal Description Techniques II, FORTE ’89, editor S. Vuong, pp. 281-296, North Holland, 1990.

[Chen76] P. P. Chen: *The Entity-Relationship Model: Toward a Unified View of Data*. ACM Transactions on Database Systems, 1(1): 9-36, ACM Press, March 1976.

[Chidamber&Kemerer91] Shyam R. Chidamber and Chris F. Kemerer: *Towards a Metrics Suite for Object-Oriented Design*. Proceedings of OOPSLA ’91, ACM SIGPLAN Notices, 26(11): 197-211, ACM Press, 1991.

[Coad&Yourdon91a] P. Coad and E Yourdon: *Object-Oriented Analysis*. Second edition, Yourdon Press/Prentice Hall, 1991.

[Coad&Yourdon91b] P. Coad and E. Yourdon: *Object-Oriented Design*. Prentice Hall, 1991.

[Codenie&al97] Wim Codenie, Koen De Hondt, Patrick Steyaert and Arlette Vercammen: *From Custom Applications to Domain-Specific Frameworks*. Communications of the ACM, 40(10): 71-77, ACM Press, 1997.

[Conradi&Westfechtel98] Reidar Conradi and Bernhard Westfechtel: *Version Models for Software Configuration Management*. ACM Computing Surveys, 30(2), ACM Press, June 1998.

[Cook&al90] William R. Cook, Walter L. Hill and Peter S. Canning: *Inheritance is not subtyping*. ACM Transactions on Programming Languages and Systems, pp. 125-135, ACM Press, 1990.

[Cook&Daniels94] Steve Cook and J. Daniels: *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994.

[Coplien&Schmidt95] James O. Coplien and Douglas C. Shmidt: *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[Cornelis97] Gerrit Cornelis: *Reuse Contracts as a Modular System in Statically Typed Object-Oriented Languages*. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1997.

[Corradini&al96a] Andrea Corradini, Ugo Montanari and F. Rossi: *Graph Processes*. In [FI96], pp. 241-265.

[Corradini&al96b] Andrea Corradini, Hartmut Ehrig, Michael Löwe, Ugo Montanari and J. Padberg: *The category of typed graph grammars and their adjunction with categories of derivations*. In [Gra96]

[Cunningham&Beck86] Ward Cunningham and Kent Beck: *A Diagram for Object-Oriented Programs*. Proceedings of OOPSLA ’86, ACM SIGPLAN Notices, 21(11): 361-367, ACM Press, 1986.

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

[DePauw&al93] Wim De Pauw, Richard Helm, Doug Kimelman and John Vlissides: *Visualising the Behaviour of Object-Oriented Systems*. Proceedings of OOPSLA ’93, ACM SIGPLAN Notices, 28(10): 326-337, ACM Press, 1993.

[D’Hondt98] Maja D’Hondt: *Managing Evolution of Changing Software Requirements*. Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1998.

[Douglass98b] Bruce Powel Douglass: *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[Duke&al91] R. Duke, P. King, G. Rose and G. Smith: *The Object-Z Specification Language*. Proceedings of TOOLS’91, pp. 465-483, Prentice-Hall, 1991.

[Ecklund&al96] E. F. Ecklund Jr., Lois M. L. Delcambre and M. J. Freiling: *Change cases: Use cases that identify future requirements*. Proceedings of OOPSLA '96, ACM SIGPLAN Notices, 31(10), pp. 342-358, ACM Press, 1996.

[Ehrig79] Hartmut Ehrig: *Introduction to the Algebraic Theory of Graph Grammars - A Survey*. In [Gra79], pp. 1-69.

[Ehrig&Habel86] Hartmut Ehrig and Annegret Habel: *Graph grammars with application conditions*. G. Rozenberg and A. Salomaa, editors, The Book of L, pp. 87-100, Springer-Verlag, 1986.

[Ehrig&al91] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski and Francesco Parisi-Presicce: *From Graph Grammars to High-Level Replacement Systems*. In [Gra91], pp. 269-291.

[Ellis95] G. Ellis: *Object-Oriented Conceptual Graphs*. Proceedings of 3rd International Conference on Conceptual Structures, LNCS and LNAI 954, pp. 144-157, Spinger-Verlag, 1995.

[Engels&Schürr95] G. Engels and Andy Schürr: *Encapsulated Hierarchical Graphs, Graph Types and Meta Types*. Joint Compugraph/Semagraph Workshop on Graph Rewriting and Computation. Electronic Notes in Theoretical Computer Science, Vol. 2, Elsevier, 1995.

[FI96] G. Engels, Hartmut Ehrig and G. Rozenberg, editors. Fundamenta Informaticae, Special Issue on Graph Transformations, 26(3,4), IOS Press, June 1996.

[Firesmith&al97] D. Firesmith, Brian Henderson-Sellers and I. Graham: *OPEN Modeling Language Reference Manual*. SIGS Books, 1997.

[Fradet&al99] P. Fradet, Daniel Le Métayer and M. Périn: *Consistency Checking for Multiple View Software Architectures*. To appear in Proceedings of ESEC '99, 1999.

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

[Gane&Sarson78] C. Gane and T. Sarson: *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, 1978.

[Garlan95] David Garlan: *Workshop Summary*. First International Workshop on Architectures of Software Systems, ACM SIGSOFT Software Engineering Notes, 20(3): 84-89, ACM Press, 1995.

[Garlan&Shaw93] David Garlan and M. Shaw: *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering. Editors V. Ambriola and G. Tortora. World Scientific Publishing Company, 1993.

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

[Ghezzi&al91] Carlo Ghezzi, Mehdi Jazayeri and D. Mandrioli: *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[Gogolla96] Martin Gogolla: *Towards Object Visualization by Conceptual Graphs*. Proceedings of 4th International Conference on Conceptual Graphs, 1996.

[Gogolla&Richters98] Martin Gogolla and Mark Richters: *On Constraints and Queries in UML*. M. Schader and A. Korthaus, editors, The Unified Modeling Language - Technical Aspects and Applications. Physica-Verlag, Heidelberg, 1998.

[Goldberg&Rubin95] Adele Goldberg and Kenneth S. Rubin: *Succeeding with Objects: Decision Frameworks for Project Management.* Addison-Wesley, 1995.

[Goldberg98] Adele Goldberg: *A reuse business model*. Software Concepts & Tools, Special issue on componentware, 19(1): 11-13, Springer-Verlag, 1998.

[Gra79] *Proceedings International Workshop on Graph Grammars and their Application to Computer Science and Biology*. LNCS 73, Springer-Verlag, 1979.

[Gra83] H. Ehrig, M. Nagl and G. Rozenberg (Eds.): *Proceedings 2nd International Workshop on Graph Grammars and their Application to Computer Science*. LNCS 153, Springer-Verlag, 1983.

[Gra87] H. Ehrig, M. Nagl, G. Rozenberg and A. Rosenfeld (Eds.): *Proceedings 3rd International Workshop on Graph Grammars and their Application to Computer Science*. LNCS 291, Springer-Verlag, 1987

[Gra91] H. Ehrig, H.-J. Kreowski and G. Rozenberg (Eds.): *Proceedings 4th International Workshop on Graph Grammars and their Application to Computer Science*. LNCS 532, Springer-Verlag, 1991.

[Gra96] *Proceedings 5th International Workshop on Graph Grammars and their Application to Computer Science*. LNCS, Springer-Verlag, 1996.

[Gra98] *Proceedings 6th International Workshop on Theory and Application of Graph Transformation*. LNCS, Springer-Verlag, 1998.

[Habel&al96] Annegret Habel, Reiko Heckel and Gabriele Taentzer: *Graph Grammars with Negative Application Conditions*. In [FI96], pp. 287-313.

[Halpin98] Terry A. Halpin: *UML Data Models From an ORM Perspective*. Parts 1-3, Journal of Conceptual Modeling, Information Conceptual Modeling, April-June 1998.

[Hamie&al98] A. Hamie, J. Howse, Stuart Kent, R. Mitchell and F. Civello: *Reflections on the OCL*. In [UML98].

[Hamie&al98b] A. Hamie, J. Howse and Stuart Kent: *Interpreting the Object Constraint Language*. Proceedings of Asia Pacific Conference in Software Engineering. IEEE Press, 1998.

[Harel&Gery96] David Harel & Eran Gery: *Executable Object Modeling with Statecharts.* Proceedings of International Conference on Software Engineering '96, pp. 246-257, IEEE Press, 1996.

[Harel87] David Harel: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, Volume 8, 1987.

[Harel88] David Harel: *On Visual Formalisms*. Communications of the ACM, May 1988, 31(5): 514-530, ACM Press, 1988.

[Heckel95] Reiko Heckel: *Algebraic Graph Transformations with Application Conditions*. Dissertation, Technische Universität Berlin, 1995.

[Heckel&al96] Reiko Heckel, Andrea Corradini, Hartmut Ehrig and Michael Löwe: *Horizontal and Vertical Structuring of Typed Graph Transformation Systems*. Mathematical Structures in Computer Science, Cambridge University Press, 1996.

[Heckel&Wagner95] Reiko Heckel and Annika Wagner: *Ensuring Consistency of Conditional Graph Grammars – A Constructive Approach*. Lecture Notes in Theoretical Computer Science 1 (1995), Elsevier Science, 1995.

[Helm&al90] Richard Helm, I. M. Holland and D. Gangopadhyay: *Contracts: specifying behavioral compositions in object-oriented systems*. Proceedings of OOPSLA/ECOOP '90, ACM SIGPLAN Notices, 25(10): 169-180, ACM Press, 1990.

[Henderson-Sellers&Bulthuis98] Brian Henderson-Sellers and Arjan Bulthuis: *Object-Oriented Metamethods*. Springer-Verlag, 1998.

[Hirsch&al99] D. Hirsch, P. Inverardi and Ugo Montanari: *Modelling Software Architectures and Styles with Graph Grammars and Constraint Solving*. Software Architecture, pp. 127-143, Kluwer Academic Publishers, 1999.

[Hofmeister&al99] C. Hofmeister, R. L. Nord and D. Soni: *Describing Software Architecture with UML*. Proceedings of Working IFIP Conference on Software Architecture, pp. 145-160, Kluwer Academic Publishers, 1999.

[Horwitz&al89] Susan Horwitz, Jan Prins and Thomas Reps: *Integrating Non-interfering Versions of Programs*. ACM Transactions on Programming Languages and Systems, 11(3): 345-387, ACM Press, July 1989.

[IBM94] IBM: *The System Object Model (SOM) and the Component Object Model (COM): A Comparison of technologies from a developer's perspective*. White Paper, IBM Corporation, 1994.

[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.

[Jacobson&al92] Ivar Jacobson, M. Christerson, Patrik Jonsson and Gunnar Övergaard: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[Jacobson&al97a] Ivar Jacobson, Martin Griss and Patrik Jonsson: *Making the Reuse Business Work*. IEEE Computer, October 1997.

[Jacobson&al97b] Ivar Jacobson, Martin Griss and Patrik Jonsson: *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[Johnson&Foote88] Ralph E. Johnson and Brian Foote: *Designing Reusable Classes*. Journal of Object-Oriented Programming, 1(2): 22-35, SIGS Publications, 1988.

[Johnson&Opdyke93] Ralph E. Johnson and William F. Opdyke: *Refactoring and Aggregation*.

[Jones90] C. B. Jones: *Systematic Software Development Using VDM*. International Series in Computer Science, Prentice Hall, 1990.

[Karlsson95] Even-André Karlsson: *Software Reuse: A Holistic Approach*. John Wiley and Sons, 1995.

[Katayama98] Takuya Katayama: *A Theoretical Framework of Software Evolution*. In [IWPSE98], pp. 1-5.

[Kim&al89] Won Kim, Nat Ballou, Hong-Tai Chau, Jorge F. Charza and Darrell Woelk: *Features of the ORION Object-Oriented Database System*. In Object-Oriented Concepts, Databases and Applications. Editors Won Kim and Frederick H. Lochovsky, pp. 251-282, ACM Press, 1989.

[Kleyn&Gingrich88] Michael F. Kleyn and Paul C. Gingrich: *GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views*. Proceedings of OOPSLA ’88, ACM SIGPLAN Notices, 23(11): 191-205, ACM Press, 1988.

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

[Lamping93] John Lamping: *Typing the Specialisation Interface*. Proceedings of OOPSLA ’93, ACM SIGPLAN Notices, 28(10): 201-214, ACM Press, 1993.

[Larman98] Craig Larman: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 1998.

[Leblang94] D. B. Leblang: *The CM challenge - Configuration Management that works.* In Configuration Management, W.F. Tichy (ed.), Trends in Software Vol. 2, pp. 1-38, Wiley, 1994.

[Lehman&Belady85] M. M. Lehman and L. A. Belady: *Program Evolution – Processes of Software Change*. Academic Press, 1985.

[Lehman98] M. M. Lehman: *Software’s Future: Managing Evolution.* IEEE Software, pp. 40-44, IEEE Press, January/February 1998.

[LeMétayer98] Daniel Le Métayer: *Describing Software Architecture Styles Using Graph Grammars*. IEEE Transactions on Software Engineering, 24(7): 521-553, IEEE Press, July 1998.

[Liao&al99] S. Y. Liao, L. S. Cheung and W. Y. Liu: *An Object-Oriented System for the Reuse of Software Design Items*. Journal of Object-Oriented Programming, 11(8): 22-28, SIGS Publications, January 1999.

[Lieberherr&al93] Karl J. Lieberherr, Ignacio Silva-Lepe and Cun Xiao: *Adaptive Object-Oriented Programming Using Graph-Based Customization*. 1993.

[Lieberherr&Patt-Shamir97] Karl J. Lieberherr and B. Patt-Shamir: *Traversals of Object Structures: specification and efficient implementation*. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, September 1997.

[Lippe&vanOosterom92] Ernst Lippe and Norbert van Oosterom: *Operation-based Merging*. Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments. ACM SIGSOFT Software Engineering Notes, 17(5): 78-87, ACM Press, 1992.

[Löwe93] Michael Löwe: *Algebraic Approach to Single-Pushout Graph Transformation*. Theoretical Computer Science 109, pp. 181-224, 1993.

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

[MacLennan87] Bruce MacLennan: *Principles of Programming Languages Second Edition*. Saunders College Publishing, 1987.

[Martin&Odell95] J. Martin and J. Odell: *Object-oriented methods: a foundation*. Prentice Hall, 1995.

[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.

[Mens98] Tom Mens: *A Basic Formalism for Systematic Software Evolution*. In [IWPSE98].

[Mens99] Kim Mens: *Managing Unanticipated Evolution of Software Architectures*. (Tentative title.) Ph. D. Dissertation, Programming Technology Lab, Vrije Universiteit Brussel, In preparation, 1999.

[Mens&al98a] Tom Mens, Carine Lucas and Patrick Steyaert: *Giving Precise Semantics to Reuse in UML*. Proceedings of the International Workshop on Precise Semantics for Software Modeling Techniques, pp. 73-89, Kyoto, Japan, April 1998. Available as Technical Report TUM-I9803, Technische Universität München.

[Mens&al99a] Tom Mens, Carine Lucas and Patrick Steyaert: *Supporting Reuse and Evolution of UML Models*. In [UML98].

[Mens&al99b] Tom Mens and Theo D'Hondt: *Automating Support for Software Evolution in UML*. Submitted to Automated Software Engineering, Kluwer Academic Publishers, 1999.

[Mens&Steyaert97] Tom Mens and Patrick Steyaert: *Incremental Design of Layered State Diagrams*. Technical Report vub-prog-tr-97-04, Vrije Universiteit Brussel, 1997.

[Mens&VanLimberghen95] Tom Mens and Marc Van Limberghen: *Self sends as primary construct: a criterion for object-oriented language design*. Technical Report vub-prog-tr-95-08, Vrije Universiteit Brussel, 1995.

[Meseguer&Montanari90] José Meseguer and Ugo Montanari: *Petri Nets are Monoids*. Information and Computation, 88: 105-155, 1990.

[Meyer92] Bertrand Meyer: *Applying Design by Contract*. IEEE Computer, 25(10): 40-51, 1992.

[Mezini97] Mira Mezini: *Maintaining the Consistency of Class Libraries During Their Evolution*. Proceedings of OOPSLA '97, ACM SIGPLAN Notices, 32(10): 1-21, ACM Press, 1997.

[Mezini&Lieberherr98] Mira Mezini and Karl Lieberherr: *Adaptive Plug-and-Play Components for Evolutionary Software Development.* Proceedings of OOPSLA '98, 33(10): 97-116, ACM Press, 1998.

[Montanari70] Ugo G. Montanari: *Separable Graphs, Planar Graphs and Web Grammars*. Inf. Contr. 16, pp. 243-267, 1970.

[Nijssen&Halpin89] G. M. Nijssen and Terry A. Halpin: *Conceptual Schema and Relational Database Design: a Fact-oriented Approach*. Prentice Hall, 1989.

[Olsen&al94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed & J. R. W. Smith: *Systems Engineering Using SDL-92*. North-Holland, 1994.

[OMG97a] Object Management Group: *UML Notation Guide*. OMG Document ad/97-08-05, Version 1.1, 1 September 1997.

[OMG97b] Object Management Group: *UML Semantics*. OMG Document ad/97-08-04, Version 1.1, 1 September 1997.

[OMG97c] Object Management Group: *Meta-Object Facility (MOF) Specification*. OMG Document ad/97-08-09, Version 1.1, 1 September 1997.

[OMG97d] Object Managament Group: *Object Constraint Language Specification*. OMG Document ad/97-08-08, Version 1.1, 1 September 1997.

[Opdyke92] William F. Opdyke: *Refactoring object-oriented frameworks*. Ph. D. Dissertation, University of Illinois at Urbana-Champaign, Technical Report UIUC-DCS-R-92-1759, 1992

[Opdyke&Johnson93] William F. Opdyke and Ralph E. Johnson: *Creating abstract superclasses by refactoring*. Proceedings of 1993 ACM Computer Science Conference, pp. 66-73, ACM Press, 1993.

[Oquendo&al89] F. Oquendo, K. Berrado, F. Gallo, R. Minot and I. Thomas: *Version Management in the PACT Integrated Software Engineering Environment*. Proceedings 2nd European Software Engineering Conference, C. Ghezzi and J. A. McDermid (eds.), LNCS 387, pp. 222-242, Springer-Verlag, 1989.

[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.

[Oreizy&Taylor98] Peyman Oreizy and Richard N. Taylor: *On the role of software architectures in run-time system reconfiguration*. Technical Report, 1998.

[Ottenstein84] K. J. Ottenstein and L. M. Ottenstein: *The Program Dependence Graph in a Software Development Environment*. ACM SIGPLAN Notices, pp. 177-184, ACM Press, May 1984.

[Page-Jones95] M. Page-Jones: *What Every Programmer Should Know About Object-Oriented Design*. Dorset-House Publishing, 1995.

[Parisi-Presicce96] Francesco Parisi-Presicce: *Transformation of Graph Grammars*. In [Gra96].

[Parnas72] D. L. Parnas: *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, 15(12): 1053-1058, ACM Press, 1972.

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

[Perry&Wolf92] Dewayne E. Perry and A. L. Wolf: *Foundations for the Study of Software Architecture*. ACM SIGSOFT Software Engineering Notes, 17(4): 40-52, 1992.

[Pfaltz&Rozenfeld69] John L. Pfaltz and Azriel Rozenfeld: *Web Grammars*. Proceedings of International Joint Conference on Artificial Intelligence, Washington, pp. 609-619, 1969.

[Pfleeger&Bohner90] S. L. Pfleeger and Shawn A. Bohner: *A Framework for Software Maintenance Metrics*. Proceedings of Conference on Software Maintenance, November 1990.

[Pirker&al98] Helfried Pirker, Roland T. Mittermeir and Dominik Rauner-Reithmayer: *Service Channels – Purpose and Tradeoffs*. 22nd Annual International Computer, Software and Applications Conference COMPSAC ’98, Vienna, Austria, 1998.

[Poulin96] J. S. Poulin: *Evolution of a Software Architecture for Management Information Systems*. Proceedings of the Second International Software Architecture Workshop (ISAW2), pp. 134-137, 1996.

[Poulovassilis&Levene94] A. Poulovassilis and M. Levene: *A Nested-Graph Model for the Representation and Manipulation of Complex Objects*. ACM Transactions on Information Systems, 12(1): 35-68, ACM 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.

[Prieto-Diaz90] R. Prieto-Diaz: *Domain Analysis: an Introduction*. ACM SIGSOFT Software Engineering Notes, 15(2), ACM Press, 1990.

[Prieto-Diaz&Freeman87] R. Prieto-Diaz and P. Freeman: *Classifying Software for Reusability*. IEEE Software, pp. 6-16, Janyary 1987.

[Reenskaug&al96] Trygve Reenskaug, Per Wold and Odd Arild Lehne: *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[Richters&Gogolla98] Mark Richters and Martin Gogolla: *On Formalizing the UML Object Constraint Language OCL*. Tok-Wang Ling, editor, Proceedings of 17th International Conference on Conceptual Modeling, LNCS, Springer-Verlag, 1998.

[Rigg&al95] W. Rigg, C. Burrows, P. Ingram: *Configuration Management Tools*. OVUM Ltd., 1995.

[Roberts&Johnson96] Don Roberts and Ralph Johnson: *Evolving Frameworks: A Pattern-language for Developing Object-oriented Frameworks*. PLoP '96 Proceedings, 1996.

[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.

[Rubin&Goldberg92] Kenneth S. Rubin and Adele Goldberg: *Object Behaviour Analysis*. Communications of the ACM, 35(9): 48-62, Special Issue on Object-Oriented Methodologies, ACM Press, September 1992.

[Rudolph&al96] Ekkart Rudolph, Jens Grabowski and Peter Graubmann: *Tutorial on Message Sequence Charts (MSC'96)*. Tutorials at First Joint Conference FORTE/PSTV'96, Kaiserslautern, Germany, October 1996.

[Rumbaugh&al91] Jim Rumbaugh, M. Blaha, W. Remerlani, F. Eddy and W. Lorensen: *Object-Oriented Modelling and Design*. Prentice Hall, 1991.

[Schlaer&Mellor92] S. Schlaer and S. J. Mellor: *Object Life Cycles - Modeling the World in States*. Yourdon Press/Prentice Hall, 1992.

[Schro&France97] M. Shroff & R. B. France: *Towards a Formalization of UML Class Structures in Z.* Proceedings of 21st Annual International Computer Software and Applications Conference, pp. 646-651, IEEE Press, 1997.

[Schürr91] Andy Schürr: *PROGRES - A VHL-language Based on Graph Grammars*. In [Gra91], pp. 641-659.

[Schürr95] Andy Schürr: *PROGRES for Beginners*. http://www-i3.informatik.rwth-aachen.de/research/progres/documentation.html, Department of Computer Science, Aachen University of Technology, 1997

[Schürr96] Andy Schürr: *Logic Based Programmed Structure Rewriting Systems*. In [FI96], pp. 363-385.

[Schwanke&al96] R. W. Schwanke, V. A. Strack and T. Werthmann-Auzinger: *Industrial Software Architecture with Gestalt*. Proceedings of IWSSD-8, pp. 176-180, IEEE Press, 1996.

[Selic&al94] Bran Selic, G. Gullekson and P. T. Ward: *Real-Time Object-Oriented Modelling*. John Wiley and Sons, 1994.

[Selic&Rumbaugh98] Bran Selic, Jim Rumbaugh: *Using UML for Modeling Complex Real-Time Systems*. White Paper, ObjecTime, April 1998.

[Shaw&Garlan96] M. Shaw and David Garlan: *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Simons&Graham98] Anthony Simons and Ian Graham: *37 Things that Don't Work in Object-Oriented Modelling with UML*. Second ECOOP Workshop on Precise Behavioural Semantics, Technical Report TUM-I9813, pp. 209-232, Technische Universität München, 1998.

[SPC93] Software Productivity Consortium: *Reuse Adoption Guidebook*. Technical Report SPC-92051-CMC, Version 02.00.05, November 1993.

[Spivey89] J. M. Spivey: *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[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.

[Szyperski98] Clemens Szyperski: *Component Software – Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.

[Tokuda&Batory98] Lance Tokuda and Don Batory: *Automating Three Modes of Evolution for Object-Oriented Software Architectures.* In 5th Conference on Object-Oriented Technologies, 1999.

[Tokuda&Batory98b] Lance Tokuda and Don Batory: *Evolving Object-Oriented Architectures with Refactorings*. Technical Report, 1999.

[UML98] Jean Bézivin and Pierre-Alain Muller, editors: *«UML»'98 - Beyond The Notation.* Selected Papers of <<UML>>'98 International Workshop, LNCS 1618, Springer-Verlag, 1999.

[Warshall62] S. Warshall: *A Theorem on Boolean Matrices*. Journal of the ACM, 9(1): 11-12, ACM Press, 1962.

[Wegner&Zdonik88] Peter Wegner and Stanley B. Zdonik: *Inheritance as an Incremental Modification Mechanism, or What Like is and Isn't Like.* Proceedings of ECOOP '88, LNCS 276, pp. 55-77, Springer-Verlag, 1988.

[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.

[Westfechtel91] Bernhard Westfechtel: *Structure-oriented Merging of Revisions of Software Documents*. Proceedings of 3rd International Workshop on Software Configuration Management, P. H. Feiler (ed.), pp. 68-79, ACM Press, 1991.

[Wiels&Easterbrook98] Virginie Wiels and Steve Easterbrook: *Management of Evolving Specifications Using Category Theory*. Proceedings of Automated Software Engineering Conference '98, pp. 12-21, IEEE Press, 1998.

[Wilde&al89] Norman Wilde, Ross Huit and S. Huitt: *Dependency Analysis Tools - Reusable Components for Software Maintenance*. Proceedings of Conference on Software Maintenance '89, pp. 126-131, IEEE Press, 1989.

[Wilde&al92] Norman Wilde and Ross Huit: *Maintenance Support for Object-Oriented Programs*. IEEE Transactions of Software Engineering, 18(12), IEEE Press, December 1992. Also in [Bohner&Arnold96a].

[Wirfs-Brock&Wilkerson89] Rebecca J. Wirfs-Brock and B. Wilkerson: *Object-oriented design: a responsibility-driven approach*. Proceedings of OOPSLA '89, pp. 71-75, SIGPLAN Notices, ACM Press, 1989.

[Wirfs-Brock&al90] Rebecca J. Wirfs-Brock, B. Wilkerson and L. Wiener: *Designing Object-Oriented Software*. Prentice Hall, 1990.

[Wood&Sommerville88] M. Wood and I. Sommerville: *An Information Retrieval System for Software Components*. Software Engineering Journal, pp. 198-207, September 1988.

VIII . APPENDICES

This chapter contains an appendix with preliminary definitions.

VIII . 1 Preliminary Definitions

In this appendix we present some basic definitions about sets, functions, relations, matrices and category theory that are used throughout this dissertation.

VIII 1.1 Sets

In this section we present some definitions and notational conventions about *sets* that will be needed in the rest of this dissertation.

- If A is a finite set, then $|A|$ denotes the **number of elements** in A.
- $\subseteq$ is used to denote **set inclusion**, while $\subset$ denotes *strict* inclusion.
- $\times$ represents the **Cartesian product** of sets, and can be used to define tuples of elements. If $a \in A$ and $b \in B$ then $(a,b) \in A\times B$.
- $\mathcal{P}(A)$ represents the **powerset** of A, i.e., the set of all possible subsets of A. Formally, $\mathcal{P}(A) = \{ B \mid B \subseteq A \}$, i.e., $B \in \mathcal{P}(A)$ iff $B \subseteq A$.

$A^1 = A$; $A^2 = A\times A$; $\forall n \in \mathbb{N}_0$: $A^n = A\times A\times\ldots\times A$ (n times); $A^+ = \cup_{n\in\mathbb{N}_0} A^n$

Definition 58: n-tuples

Sometimes, **multisets** (or **bags**) are needed instead of ordinary sets. A multiset is a set in which each element can occur more than once. Formally, this can be represented by associating a nonzero integer with each element in a set.

VIII 1.2 Functions

In this section we present some definitions and notational conventions about *functions* that will be needed in the rest of this dissertation.

If $f: A \rightarrow B$ is a function and $C \subset A$ then $g: C \rightarrow B: a \rightarrow f(a)$ is called the **restriction of *f*** to C, and is denoted by $f|_C = g$.

Definition 59: Restriction of a function

A function $f: A \rightarrow B$ does not necessarily have to be defined in each element. The **domain** of f, represented by ***dom(f)*** is the set of all elements of A in which f is defined. The **range** of f (also called *codomain* or *image*), is defined by ***ran(f)*** $= f(A) = \{ b \in B \mid \exists a \in A \text{ such that } f(a)=b \}$

$f: A \rightarrow B$ is a **partial** function if $dom(f) \subset A$. Otherwise, f is a **total** function, i.e. $dom(f) = A$

Definition 60: Partial and total function

Let $f: A \rightarrow B$ be a function. f is **injective** (or *one-to-one*) if $\forall a, b \in dom(f)$: $f(a)=f(b)$ implies $a=b$. f is **surjective** (or *onto*) if $f(A) = B$. f is **bijective** if f is both injective and surjective.

Definition 61: Injective and surjective function

Let $n \in \mathbb{N}_0$. For any function $f: A \rightarrow B_1\times B_2\times\ldots\times B_n: a \rightarrow (b_1,b_2,\ldots,b_n)$
$\forall i \in \{1,..,n\}$ we can define the i-th **projection function** $\Pi_i^f: A \rightarrow B_i: a \rightarrow b_i$

Definition 62: Projection function

As a special case of Definition 62, we can define the i-th projection function Π_i^f in the same way for a function $f: A \rightarrow B^n$. Analogously, we can also define the i-th projection function for a function $f: A \rightarrow B^+$, with the slight distinction that in this case the projection function is only a partial function, since it does not necessarily exist for all values of A.

VIII 1.3 Relations

In this section we present some definitions and notational conventions about *relations* that are needed in this dissertation. A relation is more general than a function because the mapping does not have to be unique. An element in *A* can correspond to more than one element in *B*. We will only need relations defined on finite sets.

Let *A* and *B* be two finite sets. $R \subseteq A \times B$ is called a **relation** from *A* to *B*. If $(a,b) \in R$ we usually write $a\ R\ b$. The **inverse relation** $\boldsymbol{R^{-1}} \subseteq B \times A$ is defined as $R^{-1} = \{ (b,a) \mid (a,b) \in R \}$. If $A=B$ we speak of a **binary relation** on *A*.

Definition 63: Relation and inverse relation

Since a relation can be regarded as a generalisation of a function we sometimes write $R: A \rightarrow B$ instead of $R \subseteq A \times B$. Also similar to function notation, we use $R(a)$ to denote the subset of all elements of *B* that are related to *a*: $R(a) = \{ b \in B \mid a\ R\ b \}$

Let $R \subseteq A \times B$ and $S \subseteq B \times C$ be relations on the finite sets *A*, *B* and *C*. The composition relation $S \circ R \subseteq A \times C$ is defined as follows:

$$S \circ R = \{ (a,c) \mid \exists\, b \in B \text{ such that } (a,b) \in R \text{ and } (b,c) \in S \}$$

Definition 64: Composition of relations

A useful kind of relation is a *partial order*, since it can be used to compare elements with each other.

A binary relation $R \subseteq A \times A$ is a **partial order** iff
(1) *R* is **reflexive** ($\forall a \in A$: $a\ R\ a$)
(2) *R* is **antisymmetric** ($\forall a, b \in A$: if $a\ R\ b$ and $b\ R\ a$ then $a=b$)
(3) *R* is **transitive** ($\forall a, b, c \in A$: if $a\ R\ b$ and $b\ R\ c$ then $a\ R\ c$)

Definition 65: Partial order

VIII 1.4 Matrices

Each relation $R \subseteq A \times B$ defined on finite sets *A* and *B* can also be expressed in matrix form, where the columns are enumerated with elements of *A*, and the rows are enumerated with elements of *B*. The element of a matrix *M* indexed by column $a \in A$ and row $b \in B$ is denoted by $M[a,b]$.

If $R \subseteq A \times B$ is a relation on finite sets, then the **matrix form of *R***, denoted by $M = \mathcal{M}(\boldsymbol{R})$, is defined as follows: $\forall (a,b) \in A \times B$: if $(a,b) \in R$ then $M[a,b] = 1$ else $M[a,b] = 0$.

Definition 66: Matrix representation of a relation

An interesting property is that the inverse of a relation can be found by simply taking the transpose of a matrix, i.e., the matrix obtained by swapping rows and columns. Moreover, when working with 0-1 matrices, the matrix of a composition of two relations is equal to the product of their corresponding matrices.

If $R \subseteq A \times B$ and $S \subseteq B \times C$ are relations on finite sets, then $\mathcal{M}(R)^T = \mathcal{M}(R^{-1})$ and $\mathcal{M}(S \circ R) = \mathcal{M}(S).\mathcal{M}(R)$

Property 26: Transpose matrix versus inverse relation

Note that the above property only holds when dealing with 0-1 matrices. In that case, if the product of two matrices yields an element greater than one, a 1 is put in the corresponding row and column of the product matrix.

In Definition 1 on page 44, graphs were defined as a tuple $(V, E, source: E \rightarrow V, target: E \rightarrow V)$, where *V* and *E* represent the node set and edge set, respectively. An alternative definition would be to define the edges as a binary relation over the node set, i.e., (V, E) with $E \subseteq V \times V$. The advantage of this definition is that we can use the matrix representation to represent the edges of a graph. Such a matrix is called the *adjacency matrix* of a graph. For each two nodes, it specifies whether or not they are adjacent.

If $G = (V,E)$ is a graph with a finite set of nodes V and $E \subseteq V\times V$, then $\mathcal{M}(E)$ is called the **adjacency matrix** of G.

Definition 67: Adjacency matrix of a graph

VIII 1.5 TRANSITIVE CLOSURE

If R is a binary relation, then we can compose R with itself an arbitrary number of times. The definition of this is given in a recursive way:

Let $R \subseteq A\times A$ be a binary relation on a finite set A.

$$R^1 = R; \qquad \forall n\geq 2: R^n = R^{n-1} \circ R; \qquad R^+ = \cup_{n\in\mathbb{N}_0} R^n$$

Definition 68: Transitive closure of a relation

Using this definition of transitive closure, we can determine if a particular relation (or its equivalent graph representation) contains cycles:

A binary relation $R \subseteq A\times A$ **contains a loop in *a*** if $a\ R\ a$. It is **loop-free** if $\nexists a \in A$ such that $a\ R\ a$. R is **cyclic** if R is loop-free and $\exists\, a \in A$ such that $a\ R^+\ a$. Otherwise, the relation is **acyclic**.

Definition 69: Loops and cycles

The reason why we prohibit loops in the definition of cyclic relations is that otherwise each relation that contains loops would be cyclic. This is not very useful since we are mostly interested in cycles with a path of length greater than one.

Obviously, since edges in a graph can be defined as a relation over the nodes of the graph, the above definitions can also be used to define *acyclic graphs* and the *transitive closure of a graph*. The transitive closure of a graph contains an edge between nodes if these nodes are connected to each other in the original graph by a path of length one or more.

To calculate the transitive closure T of a graph G, one can use its adjacency matrix representation, and perform a number of matrix multiplications. This idea leads to the following straightforward implementation, which was first proposed in [Warshall62]:

```
TransitiveClosure (int N, matrix &G, matrix &T)
{ int i,j,k;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      T[i,j] = G[i,j];
  for (k=0; k<N; k++)
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        if (! T[i,j]) T[i,j] = T[i,k]&&T[k,j];
}
```

Clearly, the *running time* of the algorithm is $O(N^3)$ where N is the number of nodes of the graph. We are unaware of any faster algorithms, nor of the existence of a proof that the above algorithm is optimal. However, in specific cases, such as in the case of sparse adjacency matrices, faster algorithms than $O(N^3)$ exist.

The *space* used by the algorithm is at most quadratic in the number of nodes N, i.e., $O(N^2)$, which is optimal as the resulting transitive closure can have at most a quadratic number of edges. More precisely, the number of edges in any directed graph (that is *not* a multigraph) is always less then or equal to N^2. If the graph is represented as a two-dimensional matrix or array, i.e., a list of lists, the space used is always quadratic. If the graph is represented as a hash-table of hash-tables, the used space is linear to the size of the resulting transitive closure.

If G is a graph, and T is its transitive closure, then the adjacency matrix of T is called the **reachability matrix** of G.

VIII . 2 AN INTRODUCTION TO CATEGORY THEORY

VIII 2.1 CATEGORIES

Each kind of mathematical structure (such as sets, groups, functions, relations, graphs, etcetera) can be represented in a more abstract way by using the mathematical concept of **category**, containing **objects** that have that mathematical structure, and **morphisms** between those objects that preserve this structure.

A **category** ***C*** is a directed graph where the nodes *a, b, c,* … are called ***C*-objects**, the edges *f, g, h,* … are called ***C*-morphisms**, and the following 4 axioms hold:
(A_1) $\forall$***C***-object *a*: $\exists$ identity ***C***-morphism $id_a: a \to a$
(A_2) If $f: a \to b$ and $g: b \to c$ are ***C***-morphisms, then $g \circ f: a \to c$ is a ***C***-morphism
(A_3) $\forall$***C***-morphism $f: a \to b$: $f \circ id_a = id_b \circ f = f$
(A_4) If $f: a \to b$, $g: b \to c$, $h: c \to d$ are ***C***-morphisms, then $(h \circ g) \circ f = h \circ (g \circ f)$

Definition 70: Category

Axiom **(A_1)** is usually called *reflexivity*, **(A_2)** *transitivity* and **(A_4)** *associativity*. In order to define a category ***C***, one first needs to define what ***C***-objects and ***C***-morphisms look like, then define how morphisms can be composed, and finally check if these definitions satisfy the four axioms defined above.

In the same way as we can define a subset of a set and a subgroup of a group, we can generalise this idea to define the concept of a **subcategory** of a category.

A category ***S*** is a **subcategory** of category ***C*** (denoted by $\boldsymbol{S} \subseteq \boldsymbol{C}$) if the following 4 axioms hold:
(S_1) $\forall$ ***S***-object *a*: *a* is a ***C***-object
(S_2) $\forall$ ***S***-morphism $f: a \to b$: $f: a \to b$ is a ***C***-morphism
(S_3) $\forall$ ***S***-object *a*: the ***C***-morphism $id_a: a \to a$ coincides with the ***S***-morphism $id_a: a \to a$
(S_4) $\forall$ ***S***-morphisms $f: a \to b$, $g: b \to c$: composition $g \circ f: a \to c$ in ***C*** coincides with composition $g \circ f: a \to c$ in ***S***.

Definition 71: Subcategory

In the following definitions we will assume ***C*** to be an arbitrary category.

A ***C***-morphism $f: a \to b$ is called an **isomorphism** iff $\exists$***C***-morphism $g: b \to a$ such that
$f \circ g = id_a$ and $g \circ f = id_b$

Definition 72: Isomorphism

VIII 2.2 PUSHOUTS AND PULLBACKS

As a next concept we will define the notion of **pushouts**. They are very important in the algebraic approach of graph rewriting, since they form the fundamental construction mechanism for defining graph derivations in terms of graph productions. Intuitively, pushouts specify the way part of one object is identified with part of another. They are used to glue two objects together in their corresponding parts.

The **pushout** of ***C***-morphisms $f: c \to a$ and $g: c \to b$ is the triple $(d, f^*: b \to d, g^*: a \to d)$ such that
(1) *d* is a ***C***-object, $f^*: b \to d$ and $g^*: a \to d$ are ***C***-morphisms
(2) $g^* \circ f = f^* \circ g$
(3) $\forall$***C***-object *e*: $\forall$***C***-morphisms $h_1: a \to e$, $h_2: b \to e$ with $h_1 \circ f = h_2 \circ g$: $\exists\,!$ ***C***-morphism $h: d \to e$ such that $h \circ g^* = h_1$ and $h \circ f^* = h_2$.

Definition 73: Pushout

Condition **(2)** is called *commutativity*, while condition **(3)** is referred to as the *universal property*. Graphically, this is shown on the left of Figure 51. While condition **(2)** ensures that a pushout object *d* exists, the universal property guarantees that it is *minimal*. Indeed, for any other object *e* that satisfies the same commutativity requirements, there is a unique morphism $h: d \rightarrow e$.

$$\begin{tikzcd}
c \arrow[r, "f"] \arrow[d, "g"'] & a \arrow[d, "g^*"'] \arrow[ddr, "h_1"] & \\
b \arrow[r, "f^*"] \arrow[drr, "h_2"'] & d \arrow[dr, "h"] & \\
 & & e
\end{tikzcd}
\qquad
\begin{tikzcd}
e \arrow[dr, "h"] \arrow[drr, "h_1"] \arrow[ddr, "h_2"'] & & \\
 & c \arrow[r, "g^*"] \arrow[d, "f^*"] & a \arrow[d, "f"'] \\
 & b \arrow[r, "g"] & c
\end{tikzcd}$$

Figure 51: Pushout and pullback construction

Analogously to pushout, we can define the dual notion of **pullback** by inversing all arrows.

The **pullback** of ***C***-morphisms $f: a \rightarrow c$ and $g: b \rightarrow c$ is the triple $(d, f^*: d \rightarrow b, g^*: d \rightarrow a)$ such that
(1) *d* is a ***C***-object, $f^*: d \rightarrow b$ and $g^*: d \rightarrow a$ are ***C***-morphisms
(2) $f \circ g^* = g \circ f^*$
(3) $\forall$ ***C***-object *e*: $\forall$ ***C***-morphisms $h_1: e \rightarrow a$, $h_2: e \rightarrow b$ with $f \circ h_1 = g \circ h_2$: $\exists\,!$ ***C***-morphism $h: d \rightarrow e$ such that $g^* \circ h = h_1$ and $f^* \circ h = h_2$.

Definition 74: Pullback

The schematical representation of this definition is shown on the right of Figure 51. While condition **(2)** ensures that a pullback object *d* exists, the universal property (condition **(3)**) guarantees that it is *maximal*.

The following property states that the pushout or pullback of two morphisms can be constructed in a unique way.

The pushout or pullback of two morphisms is unique modulo an isomorphism.

Property 27: Uniqueness of pushout and pullback