Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Vakgroep Informatica
Laboratorium voor Programmeerkunde

Integrative Composition of Program Generators

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

Johan Brichau

Academiejaar 2004 - 2005

Promotors: Prof. Dr. Theo D'Hondt, Prof. Dr. Kim Mens

Contents

Nederlandstalige Abstract

Recente software-ontwikkelingsmethoden zoals Model-driven Architectures (MDA), Generatief Programmeren en Product-line Engineering hebben een vernieuwde interesse gewekt in de automatische generatie van software. Deze ontwikkelingmethoden steunen immers op het gebruik van software generatoren. Zij steunen meer bepaald op generatoren die op basis van een specificatie, een complete implementatie produceren van een stuk software. Een software generator is zelf een software programma dat een bepaalde specificatie vertaalt naar een uitvoerbare implementatie. De generatie van user interfaces, parsers, data structuren,... (op basis van visuele tekeningen, grammatica's, e.d.) zijn typische en succesvolle voorbeelden. De ontwikkeling van dergelijke software generatoren is uiteraard een orde in complexiteit groter dan de enkelvoudige en manuele implementatie van de software die ze genereren. Om de complexiteit van software generatoren te kunnen bevatten worden dan ook specifieke (zgn. generatieve) programmeertalen en technieken aangewend voor hun implementatie. Deze technieken bieden vooral geschikte mechanismen aan voor het implementeren van generatoren zodanig dat die een efficient en correct stuk software genereren.

Het onderzoek dat in deze doctoraatsverhandeling beschreven wordt behandelt het bouwen van software generatoren als een modulaire compositie van andere generatoren. We stellen een generatieve techniek voor die de geschikte mechanismen aanbiedt om modulaire compositie te verwezenlijken. Dit is in tegenstelling tot bestaande generatieve technieken, waarbij generatoren vaak als alleenstaande en monolithische implementaties benaderd worden. Nochtans dienen verschillende generatoren vaak dezelfde programmastructuren en algoritmes te genereren. Dit leidt dan weer tot duplicatie in de implementatie van de verschillende generatoren. Daarenboven zijn deze gezamelijke programmastructuren vaak ook nuttig als afzonderlijk gegeneerde programmas. Het is daarom nuttig om afzonderlijke generatoren te implementeren en deze te herbruiken in de implementatie van diverse andere generatoren. Dit vereist echter een specifieke compositietechniek voor generatoren die de afzonderlijk gegenereerde programmas op een correcte wijze integreert.

Voor het bouwen van modulair composeerbare generatoren identificeren we twee verschillende soorten van compositie van generatoren: vertaalcompositie en integratieve compositie. Bij vertaalcompositie is het gegenereerde programma van de ene generator de invoerspecificatie van de andere generator. Integratieve compositie van generatoren is een compositie waarbij de geproduceerde programmas van de generatoren geintegreerd worden tot één enkel geproduceerd programma. Terwijl vertaalcompositie triviaal kan worden toegepast, vereist integratieve compositie een aangepaste compositietechniek en implementatie van de generatoren. Het is dan ook deze integratieve compositie die specifiek wordt uitgewerkt in deze verhandeling.

Bij de integratie van gegenereerde programmas is het belangrijk dat de functionaliteit van de gegeneerde programmas behouden blijft na hun integratie en ook dat hun integratie betekenisvol is. Daarom dienen bij een integratieve compositie van programmageneratoren de compositieconflicten gedetecteerd te worden en dient de integratie uitgevoerd te worden volgens een welbepaalde specificatie. Daarbij komt het vaak voor dat de gegenereerde programmas aangepast dienen te worden om de integratie te kunnen voltrekken. Deze aanpassingen kunnen gericht zijn op het structureel aanpassen van de gegenereerde programmas om de integratie te bewerkstelligen maar kunnen ook nodig zijn voor het oplossen van compositieconflicten. De generatieve techniek die in deze verhandeling geintroduceerd wordt biedt dan ook de nodige mechanismen aan voor het implementeren van generatoren die hun gegenereerd programma automatisch aanpassen aan een bepaalde integratieve compositie. We beschrijven ook de techniek van 'Generative Logic Metaprogramming' als een adequaat implementatiemedium voor integratief composeerbare generatoren. Deze techniek is een uitbreiding van de bestaande techniek voor logisch metaprogrammeren met constraint solving. De logische programmeertaal wordt daarbij aangewend voor de implementatie van een software generator. De combinatie van de logische evaluator en de constraint solver is het uitvoeringsmechanisme die de generatie en de integratieve compositie uitvoert.

Door het aanbieden van de adequate implementatiemechanismen is het bouwen van integratief composeerbare generatoren mogelijk zodanig dat deze herbruikbaar zijn in de implementatie van andere generatoren. Het composeren van herbruikbare implementaties van generatoren biedt tevens de mogelijkheid tot het implementeren van generatoren die elk een bepaalde functionaliteit genereren. Een compositie resulteert dan in een generator die een programma produceert die al deze functionaliteit bevat.

Acknowledgements

Being a researcher and a PhD. student at the Programming Technology Lab is much more than working on your research topic and graduating by writing your dissertation. In the past six years, I have not only been able to explore various interesting research topics, but I have also been involved in many teaching and project activities. I want to thank my promotor prof. dr. Theo D'Hondt for having provided me with all these opportunities and for giving me the freedom to investigate the research topics I found interesting. I especially want to thank him for promoting this dissertation and for giving me the final 'kick in the butt' to start writing!

I am also greatly indebted to prof. dr. Kim Mens, my co-promotor, who has been a great help in getting the text of this dissertation in its current state. I think he has seen some horrible draft versions of parts of this text but, nevertheless, he was still able to provide me with useful comments for improvements. Similarly, my colleague Thomas Cleenewerck has been more than a great help. He has provided me with many useful comments and ideas and he has certainly spent some valuable time in proof reading. I hope I will be able to return the favor to him very soon! I also want to thank Tom Tourwé and Wolfgang De Meuter for proof reading parts of my dissertation. Their comments have also improved this text considerably.

I also want to thank my committee members (Don Batory, Lodewijk Bergmans, Viviane Jonckers, Tom Tourwé, Dirk Vermeir) for their inspiring comments and in particular Don Batory for providing me with some very interesting and detailed comments. A special word of thanks is also more than appropriate to those people who have relieved me from some of my duties and provided me with the opportunity to focus entirely on writing my dissertation. I am greatly indebted to Maja D'Hondt, Dirk Deridder, Thomas Cleenewerck, Wim Vanderperren and Kris Gybels for assuming many of my responsibilities. This also holds for all my colleagues at the Programming Technology Lab. In order of appearance on the website: thank you Andy Kellens, Coen De Roover, Dirk Deridder, Dirk van Deun, Ellen Van Paesschen, Isabel Michiels, Jessie Dedecker, Johan Fabry, Kris Gybels, Linda Dasseville, Pascal Costanza, Peter Ebraert, Sofie Goderis, Stijn Mostinckx, Thomas Cleenewerck, Tom Van Cutsem and Wolfgang De Meuter!

Thanks to my girlfriend Annelies for putting up with me and not having fled the house during the last couple of months when I was only concerned with writing my dissertation. Last but not least, a big thank you goes to my parents who have given me the opportunity to study and who have always supported me in doing what I liked. Although my father is no longer around to thank him appropriately, he is beyond any doubt the one person who has sparked my interests in technology, computers and programming.

Chapter 1

Introduction

Program generators are designed and implemented to generate efficient and correct implementations for reusable program parts. However, program generators themselves are hardly reusable in the implementation of other program generators because they are not designed nor implemented to compose with other program generators. This dissertation presents a generative programming technique in which program generators can be composed to integrate their respective generated programs. Whenever possible, anticipated interferences in the integration of these generated programs are detected and resolved automatically.

1.1 Research Context

Implementing the same program structures and algorithms in different applications exposes application developers to ample opportunities to make the same mistakes over and over again. Therefore, it is desirable that developers reuse existing implementations in different application contexts. Such reusable parts are often more stable because of their proven uses [JGJ97]. Prominent examples of this *assembly of reusable parts* approach to software engineering are component-oriented programming [Szy98], subroutine libraries, object-oriented frameworks [JF88] and product-line architectures [BJMvH02, BLHM02]. These approaches are mostly inspired by real-world factory product-lines, where the same assembly pieces are used to build many different finished products (e.g. the same engines, wheels, batteries and so on... are all used in different types of cars).

In today's software development, reuse is most often achieved through the use of subroutine and component libraries. These libraries provide adequate data structures, algorithms, components, etc... that can be reused in many different applications. The problem with such a library is that the library components offer a fixed implementation and behavior. Adaptations to the provided behavior, that cannot be addressed through parameterization and specialisation, necessitate changes to the internal implementation and, consequently, require a deep understanding of the implementation. This is in contrast to the frequent need for such variations in the behavior of reusable library parts. Users of a library part often require that one or more features are included in the library part's functionality. To cope with this need, library developers are forced to provide multiple versions of each library part that implement one or more desired adaptations. The Booch library of data structures [Boo87] is a good and often cited example of a library that provides an implementation for 17 possible data structures (stacks, lists, queues, etc...) along 11 general features that can be included in each of these data structures [Big98]. For the queue data structure alone, there are 26 meaningful implementations that each provide a different combination of features [Big98]. As the number of desired features increases, this inevitably leads to a combinatorial explosion of the number of parts that need to be implemented in the library. This observation is referred to as the *library scaling problem* [Big94, BSST93, Big98] and leads to serious maintenance and evolution problems. The primary cause for this problem is that conventional programming techniques fail in providing adequate abstractions to modularize the implementation of the features such that each possible library part can be built as a composition of modules. As a result, the implementation of the same feature is duplicated in various versions of the library part. Moreover, particular combinations often require subtle changes to the implementation of the individual features. This further complicates the maintenance of the individual features included in the different versions. Although object-oriented languages already provide adequate support for the modular implementation of some feature-variations through frameworks, the possible variations and separation of functionalities remain limited. For example, the static structure of a framework remains fixed and cannot be adapted and the specialization of original framework behavior can only be expressed by means of inheritance or association. Last but not least, not every possible feature can be adapted through parameterization because it cannot always be represented as a first-class implementation element. These limited possibilities of adaptation in object-oriented languages have also been addressed in research on aspect-oriented software development [KLM$^+$97, SB00, TOHJ99]. In aspect-oriented languages, some additional linguistic abstractions were introduced to modularize the implementation of adaptations to methods and classes that cannot be implemented in a scalable way using traditional object-oriented languages. This also illustrates the need for adaptations to program parts that cannot be expressed using the linguistic features available in a particular programming language.

1.1.1 Program Generation

Generative approaches to software engineering [Cza98, Cle88, SB00] offer a different approach for the implementation of reusable parts and tackle the library scaling problem by a mechanized process that produces the different implementations of a library part in a systematic way [SB00].

Although program generators have been mentioned as long as 25 years ago [Joh79, Nei80] (see also [SZD00]), the interest in program generation has grown rapidly over the recent years by the advent of software development paradigms such as product-line engineering [BJMvH02, BLHM02], generative programming [Cza98] and model-driven architectures (MDA) [Gro]. Program generators are at the very heart of these development paradigms, where they are used to build completely functional implementations of (reusable) program parts. The generated implementation of these reusable program parts requires no (manual) modifications by the developer. In essence, the generated program part can be used in the implementation of an application as a black-box reusable program part. This is in contrast to the use of program generation in CASE (Computer-Aided Software Engineering) tools, where only a part of an implementation is generated and where manual completion of the generated code is required (e.g. in the production of code skeletons from UML diagrams). This latter use of program generation is primarily targeted to increase programming productivity and not at the generation of complete programs.

Over the years, many program generators were developed, some of which have specifically illustrated the usefulness of program generation. Perhaps Lex and Yacc [LMB] are the best known program generators. They can produce a parser for many language grammars, specified in an extended BNF language. The generation of parsers is an adequate application of program generation because it is impossible to provide a library that contains all possible parsers. This is because the variation in functionality between each parser cannot be captured through parameterization or object-oriented specialisation without running into the library scaling problem or a very inefficient implementation. Another prominent example of a program generator is a user-interface generator. A UI generator produces code to construct a user-interface that is specified by the developer (e.g. in a visual builder tool). A good example of such a generator is included in the Visualworks Smalltalk development environment. Another frequently used program part is a data container that allows to store a collection of values. DiSTiL [SB97] is an example generator that produces a data container according to a set of desired features that can be included in the implementation of the data container. All these generators produce completely functional implementations that can be used in the development of an application. This is also the kind of program generators that is considered in this dissertation, which is discussed next.

1.1.2 Generative Programming

Any program that produces another program as output can be called a program generator. This includes compilers for programming languages as well as code-skeleton generators for UML models. However, code-skeleton generators are CASE tools that assist a developer in implementing a program but they do not implement a complete program. More specifically, the generated program requires further completion by the developer to implement its functionality. In contrast, compilers produce an entire program that should not be modified by the developer. The generated program of the compiler should thus be considered as a 'black box' because we do not require any knowledge on its implementation details and should not modify them. Furthermore, modifying a generated program produced by a compiler is particularly difficult since we can easily break the generated functionality. Likewise, program generators that are considered in the *generative programming* paradigm [Cza98] aim for an automated mapping of a high-level description to a completely functional low-level implementation. This dissertation considers program generators in the context of generative programming and consequently considers program generators that produce a completely functional program part. Such program generators accept a specification of features and produce the corresponding program part by generating an implementation that only contains code for the requested features.

The possible implementations that can be generated are fixed by the program generator. As a consequence, a single program generator represents an entire (and closed) family of library parts: each generated part implements one or more of the possible features offered by the generator. Because this specification of requested features is often expressed in a high-level language, program generators can also be seen as compilers for domain-specific languages [vKV00, SB00]. However, a program generator is usually different from a traditional compiler because it produces a program written in a high-level programming language instead of machine code. Most of the time, this is because it alleviates the effort of writing the generator and delegates the further translation of high-level programs to bytecode to the compiler of the high-level language. Of course, the development of a generator is more difficult than the (manual) development of the individual programs that can be generated. But for highly reusable domains, the development of a generator and the design of its corresponding domain-specific language is often worth the extra effort. Furthermore, many development techniques and technologies exist that support the implementation of program generators. While program generators can be implemented using any programming language, particular languages offer linguistic features and abstractions that are more appropriate for the development of program generators [vWV03]. These languages can be referred to as *generative programming languages*. Although these generative programming languages and techniques already provide a number of advantages to the developer of a program generator, no contemporary technique provides abstractions to implement sufficiently modular composable program generators, which is the topic of this dissertation.

1.2 Problem Statement: Composability of Program Generators

Although program generators are a powerful and scalable implementation technique for a family of reusable program parts, reuse of program generators themselves in the implementation of other program generators is more problematic. This is because in contemporary generative programming techniques, a program generator is considered in isolation. A generator produces a single standalone program that is implemented in one or more encapsulated modules (available in the program's implementation language). The application developer can use the generated program in his application by accessing the well-defined interface of the generated program. Conversely, program generators are not designed nor implemented to be composed. More specifically, generators cannot be composed such that their generated programs are correctly integrated into a single generated program. A correct integration of generated programs is a composition of the generated programs that:

- prevents undesired interferences between the generated programs such that their functionality is not broken.
- implements the required interactions to achieve a combined functionality of the generated programs.

A correct integration is trivially possible if the generated programs are implemented using encapsulated modules that merely require interaction through their respective interfaces. However, many integrations and the corresponding interactions between generated programs are hard or even impossible to achieve using the composition mechanisms available in the program's implementation language. This is especially true for the integration with program parts that cannot be implemented in encapsulated modules. Such crosscutting program parts are also commonly referred to as *crosscutting concerns* [KLM$^+$97, TOHJ99]. The implementation of a crosscutting concern is distributed over multiple program modules and requires an *invasive integration* in other (generated) programs. An invasive integration means that the generated program parts are inserted in the internal implementation of a generated program. Such an invasive integration clearly modifies the internal implementation and can consequently cause interferences that break the functionality of the generated programs.

In contemporary generative techniques, an invasive integration of separately generated programs requires that the developer builds an entirely new generator or that he accomplishes the integration manually. However, manual integration of generated programs is most undesirable. In a manual invasive integration, the developer needs to understand the internal implementation details of both generated programs to establish the desired interactions and to prevent the undesired interferences in the integration. Consequently, a manual integration requires profound knowledge on the internal implementation details of the generated programs. This cripples the advantage of using a program generator or even the use of a reusable program part.

We conclude that the lack of composability of program generators harms their reusability. Developers are forced to implement the generation of similar or identical program parts in each generator instead of reusing an existing generator that produces those program parts. Some existing generative programming technologies already provide a modularization mechanism for the implementation of program generators but the modules are almost always tightly coupled. This lack of genericity cripples the reusability of the modules outside their original generator context [Big00].

1.3 Thesis Statement

In this dissertation, we introduce a generative programming technique that supports the development of *integrative composable* generators. Integrative composition of generators results in a composed generator that produces an (invasive) integration of the respectively generated programs. In an integrative composition, the required interactions between the generated programs can be implemented while all anticipated undesired interferences are automatically prevented. An integrative composition of program generators does not require a profound knowledge on the entire internal implementation of a generated program, nor does it require knowledge on the internal implementation of the program generator. However, it does require that the integrative composable generators are designed and implemented for integrative composition.

Integrative composable generators permit us to modularize the generation of individual concerns in separate program generators. An integrative composition of such generators produces a program that implements all concerns. In other words, integrative composable program generators permit the modular implementation of program generators. Such modularization improves the maintainability and evolvability of a program generator. It also allows to reuse a program generator, that produces an implementation for a particular concern, in the development of other generators that also require to generate that concern.

1.4 Approach of the Dissertation

1.4.1 Integrative Composable Program Generators

An integrative composition of program generators conflicts with the desirable black-box property of their generated programs. Some knowledge on the internal implementation of the generated programs is required to achieve an integration of these generated programs. More specifically, to specify an invasive integration of generated programs, we need to be able to specify at what specific locations in the generated program the integration needs to occur. Furthermore, an invasive integration also requires us to actually modify the internal implementation of the generated program at those locations. Last but not least, the generated programs may require to be adapted at various other locations to achieve a correct invasive integration with the desired interactions and without the undesired interferences. Therefore, integrative composable program generators break the black-box property of their generated programs and provide a controlled access to the internal implementation of these programs. This access is provided through an *integrative composition interface* defined by the program generator. However, while some internal implementation details of the generated program are exposed for integrative composition purposes, many more internal implementation details remain hidden. It is imperative that these internal implementation details are not broken as a consequence of the integrative composition. Therefore, an integrative composition is governed by a *composition conflict detection* mechanism that prevents detectable undesired interferences in the integration of the generated programs. Furthermore, to accommodate particular integrations and to resolve some of the possible composition conflicts, the generated program of a generator can be automatically adapted by the program generator itself. A program generator accomplishes this by providing alternative implementations for its generated program. These alternative implementations are driven by the possible composition conflicts and the specification of an integrative composition. Therefore, the implementation of integrative composable program generators also requires an appropriate *generative programming language* that provides support for the generation of alternative implementations of the generated program. We describe the importance of the generative programming language and the composition conflict and resolution mechanism in more detail in the following paragraphs.

Generative Logic Metaprogramming

The technique for implementing integrative composable generators presented in this dissertation is independent of a particular generative programming language. In essence, we present an overall architecture for the implementation of generators to enable integrative composition. However, to validate the concepts of the proposed technique, we present the technique of logic metaprogramming [Wuy01, DVMW00, Vol98, MMW02] as an appropriate generative programming language for the implementation of integrative composable program generators. In particular, the logic metaprogramming language offers linguistic support that is appropriate for the implementation of *integrative variabilities*. Integrative variabilities are the adaptations that are required in the implementation of a generated program to integrate with another generated program. This includes the necessary adaptations to implement the desired interactions in the integrated generated programs and the adaptations to resolve particular undesired interferences.

Composition Conflict Detection and Resolution

An integrative composition often requires that interactions occur in the integration of the generated programs. However, other interactions need to be prevented because they might break the functionality of the generated programs. These latter kind of interactions are more appropriately referred to as undesired interferences or *composition conflicts*. Although it is impossible to detect all possible composition conflicts in an integrative composition, it is guaranteed that all anticipated composition conflicts in a particular programming language are detected. The composition conflict detection mechanism is built into the proposed generative programming technique. This mechanism verifies all integrative compositions for the occurrence of declared composition conflicts. These composition conflicts are declared for each programming language that can be used as an output language for generators built in the technique.

Once a composition fails because of a detected conflict, the composition conflict needs to be resolved. Integrative composable program generators can automatically resolve certain composition conflicts by providing alternative implementations for their generated programs. An alternative implementation provides the same functionality in the generated program using an implementation that circumvents the composition conflict. These alternative implementations are implemented by the developer of an integrative composable generator. The automatic conflict resolution mechanism prevents that the developer who composes the generators needs to adapt a generated program (or even a program generator) to resolve the composition conflicts.

In this dissertation, we implement a composition conflict detection and resolution mechanism using a system that combines a constraint checker with logic metaprogramming. The constraint checker provides the core mechanism for the composition conflict detection and resolution mechanism. The possible composition conflicts can be declared as constraints in the logic metaprogramming language. These constraints are verified and enforced by the constraint checker. The composition conflicts are resolved automatically because the constraint checker enforces the selection of appropriate alternative generated programs that do not violate the constraints. This also means that the actual resolution to composition conflicts is implemented by the program generators themselves. This conflict detection and resolution mechanism is also language-independent and can be used for program generators with various output languages. For this purpose, the system can be configured with language definitions that specify the possible composition conflicts in each language. Consequently, we can detect and resolve composition conflicts in the integration of programs in various languages.

1.4.2 Domain-specific Integrative Compositions

An integrative composition can be specified for program generators that share the same output language. These output languages can be situated at many different levels of abstraction. This is because a program generator does not always produce a low-level implementation directly. Instead, a generator can produce code in a domain-specific language that needs to be further translated into executable (low-level) code. This translation process is done by another generator that accepts a program in that domain-specific language. The resulting stepwise translation from high-level specifications to low-level code through various domain-specific languages provides an opportunity for integrative composition in these domain-specific languages. What is even more important is that an integrative composition at a domain-specific level also provides the opportunity to detect and resolve domain-specific composition conflicts.

The technique for integrative composition that is developed in this dissertation is language independent and can be applied easily to integrative composition in domain-specific languages. Furthermore, the technique provides specific support for the integrative composition of generators that are composed of several other generators that perform a stepwise translation of the program through various languages. An integrative composition that occurs in a common domain-specific language used by both generators ensures that all anticipated domain-specific composition conflicts, as well as all conflicts in more low-level languages are automatically detected and prevented.

1.4.3 Feature Description Language

Although domain-specific integrative compositions are particularly desirable, the design and implementation of domain-specific languages requires a tremendous effort. Therefore, we also provide an alternative approach that does not allow domain-specific integrative compositions but does allow to detect and resolve higher-level (domain-specific) composition conflicts. This approach is based on the use of a *Feature Description Language*. This language can be used as an input language by many different generators and allows to describe more semantic properties about the generated programs.

1.5 Contributions of the Dissertation

The major contributions of the research in this dissertation are the following:

Integrative Composition of Program Generators We identify the need to build program generators that can be composed to integrate their generated programs. We develop an architecture and describe the technique to build program generators that can be composed with the intention of integrating their generated programs. This composition is referred to as *integrative composition*.

Composition Conflict Detection and Resolution We describe how to detect anticipated composition conflicts in the integration of generated programs and provide a mechanism for the automatic resolution of these composition conflicts whenever possible.

Generative Logic Metaprogramming We present the technique of logic metaprogramming as an appropriate generative programming language for the implementation of integrative composable program generators.

1.6 Perspectives

Integrative composition of program generators presents opportunities in the domain of *model-driven architecture* (MDA) and *aspect-oriented programming*.

1.6.1 Model-Driven Architecture

The Model-Driven Architecture (MDA) [Gro] is an initiative by the Object Management Group (OMG) to define an approach to software development based on modeling and automated mapping of models to implementations. The vision of MDA is that platform-independent models (PIMs) are automatically mapped onto more platform-specific models (PSMs).

Generative programming techniques are a likely approach for transforming the PIMs into PSMs and further into code. There even exist quite a number of transformation techniques that are developed and used specifically in the context of MDA [CH03]. The transformation of the PIM into the PSM requires that platform-specific details are injected into the PIM. This is specifically illustrated by the automated mapping of what MDA refers to as pervasive services. Pervasive services are, for example, transactions, security and directory services. These are typical examples of program elements that will ultimately crosscut the model or the executable code [DD02].

The realization of automated mappings will thus ultimately require the integration of the generated code for the application and the generated code for the pervasive services. Building the transformations as integrative composable program generators presents an interesting opportunity for the technique developed in this dissertation.

1.6.2 Composable Aspect-specific Languages

A particular application of composable program generators is the building of composable aspect weavers. Aspect weavers are compilers for aspect languages and therefore they are program generators. An aspect weaver takes an aspect description and integrates that aspect's implementation in the base program. The languages that are used to describe the aspects are called aspect languages. On the one hand, some aspect languages offer a high-level domain-specific language to describe the aspect's functionality (e.g. RG [MKL97], COOL [Lop97]). On the other hand, because there can be many kinds of aspects in a single program, more general-purpose aspect languages were developed (e.g. AspectJ [KHH$^+$01] and HyperJ [OT99]). Aspects written in a domain-specific language are easier to read and write because they are described at the problem domain level. However, a domain-specific aspect language can be used to express a specific kind of aspects only, while a general-purpose aspect language can express a wider range of aspects. The downside of aspects written in a general-purpose aspect language is that they lose all advantages associated with domain-specific languages and consequently are more difficult to write and understand.

Building composable aspect weavers for domain-specific aspect languages reconciles the advantages of domain-specific aspect languages with the ability to implement multiple kinds of aspect in a single application. Using contemporary generative techniques, building a new aspect language would require to change the entire aspect weaver. Modularly composable aspect weavers, on the other hand, can be composed to build a new aspect weaver that weaves multiple kinds of aspects, expressed in different aspect-specific languages.

1.7 Outline of the Dissertation

Chapter 2 provides an overview of contemporary generative programming techniques and technologies.

Chapter 3 analyses the modular composition of program generators and establishes the need for integrative composition of program generators.

Chapter 4 explains the overall generative programming technique for integrative composable program generators without delving into concrete implementation details.

Chapter 5 provides an implementation of the technique introduced in chapter 4 using generative logic metaprogramming.

Chapter 6 describes the Feature Description Language and how it is implemented in the generative logic metaprogramming technique.

Chapter 7 validates the approach by implementing a library of integrative composable program generators.

Chapter 8 concludes the dissertation and describes future work.

Chapter 2

Program Generation

Program generation automates a part of the building process of a software application. Many different techniques have been developed over the past few decades, covering a wide spectrum of applications and ranging from partial code generation based on domain models to domain-specific languages, software product lines and model-driven architectures. In this chapter, we provide an overview of program generation techniques in the context of the generative programming paradigm and describe a number of representative technologies.

2.1 Introduction

Most computer software is built to automate labour-intensive, repetitive and complex tasks. Automatisation of these tasks often yields advantages in increased productivity and reliability. The initial investment to develop the software pays off because the activity it automates needs to be executed frequently and reliably. However, such repetitive tasks are also very common in the development of the software itself. Developers often need to write identical or similar pieces of code in different software applications. This also exposes those developers to ample opportunities to make the same mistakes over and over again. *Program generation* offers a solution by automating the implementation of such reusable program parts. Generation of program parts or the generation of entire software applications introduces automatisation in the software development process itself. A *program generator* is a software program that implements an automated programmer. It is implemented once and can be applied to generate the same or similar programs many times with the same reliability. Like any other reuse technique, a program generator is most useful to automate the building of frequently-used programs or program parts. But when compared to more traditional implementation techniques for reusable parts (such as components, frameworks, etc. . .), a program generator is a scalable implementation technique for an entire set of *similar* reusable program parts.

Program generation is, of course, a broad concept in computer science. It is at the heart of a broad range of techniques, tools and development paradigms in software engineering. The most well known program generators for software developers are probably compilers [ASU86]. A compiler transforms a program written in a high-level programming language into a semantically identical program in low-level bytecode. A totally different kind of program generation can be found in integrated development environments that generate code skeletons based on UML design models[FS00]. The advent of generative programming [Cza98], product-line architectures [BJMvH02, BLHM02] and MDA [Gro] has further boosted interest and research in program generators. Depending on the context, program generators produce entire applications, components, classes, methods, code skeletons, etc. . . . Therefore, they are often referred to with different names such as application generators, component generators, code generators, software generators, etc. . . . Furthermore, the automatic derivation of algorithms from a semantic specification is also referred to as program generation or program synthesis [Smi90, SJ95]. However, the required implementation techniques for the implementation of program synthesizers, application generators and compilers are very different. Therefore, we first define the kind of program generators that are considered in this dissertation.

2.2 Definition

Program generation is the process of producing code automatically by a program generator [SZD00]. In contrast with compilers, most program generators produce a program in a high-level language instead of bytecode. Of course, in essence, any program that produces any program code as output can be called a program generator. Furthermore, most program generators produce their program based on some input specification, commonly provided by the developer. This input specification is often a program itself, which means that such program generators are actually program transformers: they transform an input program into an output program. Program transformation can be defined as follows [Unk]:

> The act of transforming one program into another.

In fact, many program generators are program transformers, but for program generators, the input language is very different from the output language. As such, program transformations such as refactorings [Fow99] cannot be considered as program generation. In many cases, program generators are also considered as compilers for domain-specific languages [vKV00]. This means that the input specification is written in a domain-specific language, which is a high-level language specifically designed to express abstractions applicable in a certain domain. The compiler for the domain-specific language is thus actually a program generator, that generates an implementation of the domain-specific program in an executable language.

In the context of this dissertation, we focus on program generators in the context of *generative programming* and product-line architectures. Czarnecki [Cza98] defined generative programming and program generators as follows:

> **Generative programming** is a software-engineering paradigm based on modeling software families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.
>
> **A program generator** is a program that takes a higher level specification of a piece of software and produces its implementation. The piece of software could be a large software system, a component, a class, a procedure, and so on.

In this dissertation we simply call the program part produced by a generator a 'program'. The definition of generative programming implicitly assumes that the generated *end-product* is a completely functional software component. This is important as it means that we consider that a generated program implements its complete functionality such that the developer is not required to modify the generated program after it has been generated. This specifically rules out skeleton code generators. Furthermore, this definition points out that program generation specifically targets the construction of a *customized* and *optimized* program. This is a particular advantage of program generation when it is compared with other implementation techniques for reusable progams, which we discuss in the following section. The *elementary, reusable implementation components* are the pieces of the program that are used by the generator to construct the resulting program. They can be implemented in text files, patterns, templates, transformations, ... The *configuration knowledge* mainly consists of the *higher level specification* that is provided to the program generator. Other parts of the configuration knowledge are contained in the generator itself and specify rules and constraints on valid compositions of the elementary implementation components and perhaps even other knowledge that the generator may use to generate a correct and efficient program.

Another definition of generative programming can be found on the webpage of the GCSE working group [Cza]:

> The goal of generative and component-based software engineering is

> to increase the productivity, quality, and time-to-market in software development thanks to the deployment of both standard componentry and production automation. One important paradigm shift implied here is to build software systems from standard componentry rather than "reinventing the wheel" each time. This requires thinking in terms of system families rather than single systems. Another important paradigm shift is to replace manual search, adaptation, and assembly of components with the automatic generation of needed components on demand. Generative and component-based software engineering seeks to integrate domain engineering approaches, component-based approaches, and generative approaches.

We can conclude by saying that in the rest of this dissertation, program generators are software programs that, given a high level specification, produce an efficient implementation of a software program that may not be adapted by the developer by manually implementing or changing parts of the generated program code.

2.3 Generation as a Reuse Technology

Program generators are particularly useful in the context of software reuse [BST^{+}94a]. Subroutine libraries, object-oriented frameworks and component libraries are the most common solutions used by developers today to accomplish reuse of frequently needed program parts. However, subroutine and component libraries offer reusable parts that have a fixed behavior that cannot be changed by the application developer. This is in contrast with the frequent need to make slight variations to the behavior of some parts [Nei80]. The library developer can anticipate this need and provide a number of different versions for each part. However, this frequently leads to a large library, containing many different versions of the same reusable part. The Booch library of data structures [Boo87] is a good and often cited example of a library that provides an implementation for 17 possible data structures (stacks, lists, queues, etc...) along with 11 general features that can be included in each of these data structures [Big98]. For the queue data structure alone, there are 26 meaningful implementations that each provide a different combination of features [Big98]. Maintaining and evolving such a library becomes quite a difficult task. This is because for each variation, a complete implementation has to be written, often leading to code duplication. Because of this code duplication, this library scaling problem [Big94, BSST93] hampers maintainability of the library. Although object-oriented frameworks have also proven their usefulness in the implementation of customizable programs, they only provide a limited solution to the scaling problem. In essence, object-oriented frameworks allow customization through inheritance and parameterization. Through inheritance, the existing methods can be extended with new behavior but the extension cannot effectively change the existing implementation in the superclass. Last but not least, in many cases, it is even impossible to package a frequently used program part as a component, a subroutine or a framework with an efficient implementation. Object-oriented design patterns [GHJV95], for example, implementations of tree structures, parsers, collaboration schemes such as publish-subscribe, etc... cannot be implemented efficiently by means of frameworks or fixed library components in all their possible versions without requiring a lot of manual customizations by the application developer.

Generative approaches come to the rescue and provide a powerful technique for the efficient implementation of frequently used program parts in a scalable way. The library provider implements a program generator that is able to generate all different versions of the reusable part. The program generator produces the implementation as it is described by an input specification. A generator actually represents an entire (and closed) family of reusable programs: each possible generated program implements a different combination of functionalities offered by the generator. A widely known example of a program generator is a parser generator. It is impossible to build a library that contains a parser for each possible programming language and a framework to build parsers can only provide some commonly used structures, still leaving much of the implementation process to the developer. The only scalable solution for an efficient implementation is to build a program generator that generates a parser for a given (programming) language as described by the developer. Parser generators such as Lex & Yacc [LMB] and SmaCC [BR] have proven the usefulness of program generation. Using these tools, a parser can be generated for many programming languages, given a grammar specification (in the BNF domain-specific language for grammars) as input to the generator. More advanced software generators have also been built for the domain of data structures. For example, DiSTiL [SB97] is a generator for data structures. The output of the generator is an implementation of a data container.

2.4 Generative Programming Techniques

2.4.1 Classification

Building a program generator is quite a complex task. Therefore many different *generative programming* techniques were developed to improve and alleviate the effort of the implementation of a program generator. Based on the implementation technique used to build a program generator, we can identify four major kinds of program generators:

Ad-hoc Many program generators are developed using standard compiler implementation techniques and tools. Most of their implementation is written in a general-purpose programming language and the generator is a stand-alone executable.

Some prominent examples are, of course, language compilers but also parser generators such as Lex and Yacc.

Metaprogramming Using metaprogramming libraries or reflective programming language facilities, we can also build program generators. C macro facilities are a weak version of this technology, but Scheme and Lisp macros are a lot more powerful in this context. The Smalltalk and OpenC++ metaprogramming facilities are also an example technology we can use to build program generators.

Transformational Program transformation systems are a powerful technique to implement program generators. In this setting, a generator is implemented as a set of program transformation rules, which are applied to an internal representation of the input specification. Each transformation rule rewrites a small part of the program and the subsequent application of these transformation rules ultimately results in the generated output program. Some program transformation systems that are well known in the context of program generation are Draco and ASF+SDF.

Compositional Programs can also be generated by glueing smaller program parts together. These program generators are implemented using a composition system that composes generic program fragments. These program fragments are parameterized such that they can be customized to fit in a particular composition. A program generator generates an output program by selecting the appropriate program fragments, based on the input specification, and assembles them to produce the generated output program. The GenVoca system is a prominent example of this kind of program generators.

2.4.2 Commonalities and Variabilities

A domain analysis precedes the development of a program generator. In such a domain analysis, the commonalities and variabilities of the different possible generated programs are identified. The commonalities are those functionalities of the generated programs that need to be included in each generated program. The variabilities are the functionalities that can be included, excluded or adapted in each of the generated programs. Deciding what is variable and what is common (or invariant) corresponds to deciding what parameters a subroutine needs to make it reusable. In program generators, the variabilities determine what a user of a generator can specify in an input specification.

In the implementation of a program generator, the developer has to implement the generation of the common and variable functionalities of the generated programs.

Many different generative techniques exist. We will now describe each of these techniques more in detail. For each technique, we will describe some of the most prominent concrete technologies.

2.5 Ad hoc Generators

Many generators are stand-alone programs that are implemented in a general-purpose programming language. We call this kind of generators *ad-hoc generators* because they have been built without the use of a specific generator infrastructure. Building such ad-hoc generators closely resembles the development of any software application. They are, in fact, written as normal software applications that happen to produce program code as output. A particularity is that many ad-hoc generators are developed using standard compiler implementation techniques and tools. Therefore, we shortly introduce the common implementation architecture of a compiler. More information on compiler implementation techniques can be found in [ASU86].

In figure 2.1, the standard internal form of a compiler is shown. The front-end of a compiler accepts the input specification and produces an internal representation for it. A translator manipulates this internal representation and converts it into a representation of the resulting program. Finally, a back-end produces the resulting program in the desired output format. The front-end of commonly known compilers is a scanner and a parser that produce an internal parsetree representation of the program. The translator is a machine-code generator that converts the tree into the a representation of the compiled program and possibly performs some optimizations. The back-end of most compilers outputs the real machine code into a file on the disk.

Implementing ad-hoc generators requires an enormous amount of effort. Besides the use of tools such as parser and scanner generators, it is a completely manual process. Developers will have to design and implement the internal representation and the translator completely from scratch. Moreover, ad-hoc generators provide no interoperability as each ad-hoc generator uses his own internal representation and input notations. An ad-hoc generator is a complete black box, thereby completely compromising its composability and interoperability with other generators. The extensibility and reusability of an ad-hoc generator is also very low, as this not only requires access to the source code but also a deep and thorough understanding of it.

Of course, almost all compilers are examples of ad-hoc program generators. Other quite well known examples are the parser generator tools Lex & Yacc [LMB] and SmaCC (parser generator in Smalltalk) [BR] .

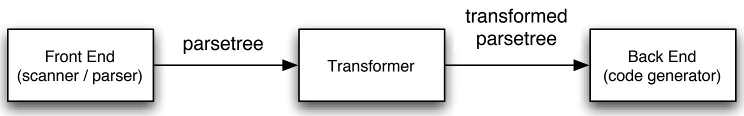


Figure 2.1: Traditional Compiler Architecture.

2.6 Metaprogramming Languages and Libraries

Instead of building a program generator 'from scratch', it is easier to make use of the metaprogramming facilities offered by a general-purpose programming language or by a metaprogramming library for the language. The Smalltalk Meta-Object Protocol (MOP) [Riv96, GR83], OpenJava [TCKI00], OpenC++ [Chi95] and the .net CodeDOM [Mic] are typical examples of metaprogramming libraries available in a general-purpose programming language. They are also referred to as API-based program generators [Voe] because the library provides an interface that can be used to perform program generation. Another kind of metaprogramming is through syntax-extension mechanisms such as macros, integrated in many programming languages such as C, Lisp and Scheme, which we will discuss in the end of this section. There are, of course, differences in possibilities in each of the metaprogramming libraries and facilities we mentioned. The Smalltalk MOP, for example, allows full runtime reflection as opposed to the OpenC++ and OpenJava libraries that only allow for compile-time metaprogramming. The .net CodeDOM supports metaprogramming for multiple .net languages as opposed to OpenJava that is specifically focused on Java. Nevertheless, in the context of program generation, we are only concerned with facilities available for program generation, either at compile-time or at runtime. This means that we are interested in how each library implements abstractions to represent a program and what operations are available to build and manipulate a program.

All metaprogramming libraries or reflection protocols offer an implementation to represent and manipulate a program. Once again, the internal representation of such a program is commonly an abstract syntaxtree. A program generator built with the use of a metaprogramming library is a program that uses the library to construct an internal representation of the program to be generated. This renders the difference between this kind of generators and the previously described ad-hoc generators rather small: i.e. both kinds use a general-purpose language to implement a program generator. The only advantage over ad-hoc generators is the reduced implementation effort and the reusability due to the common internal representation. A shared internal representation facilitates reuse of existing generators in the implementation of a new generator.

In the Smalltalk MOP, a Smalltalk class is represented by an object instance of the class `Metaclass`, which implements methods to allow various manipulations. We can, for example, add or remove methods, instance variables, etc.... Each method is also represented as an object instance of the class `CompiledMethod` that also supports various manipulations through methods. Furthermore, there are classes to represent each Smalltalk language construct in the parsetree of a method. Through these MOP facilities, we can manipulate existing programs as well as create new programs. It is not our intention to describe the entire Smalltalk MOP here or not even all facilities for static metaprogramming. The interested reader is therefore referred to [Riv96, GR83]. The approach taken by the other metaprogramming libraries (OpenJava, OpenC++,...) is very similar. Each kind of abstract syntax element is represented by a separate class that implements various manipulation methods. To illustrate the implementation of a program generator through metaprogramming, we include an example taken from the online manual of OpenJava in figure 2.2. This program generator automatically implements empty methods in a class according to the interfaces that the class implements. The execution of the generator starts with the `translateDefinition()` method. In this method, all inherited methods are retrieved from the class definition, available through the `this` variable. For each inherited method that is abstract, is not overridden in the class itself and has a void return type, we generate an empty implementation on the class using the `makeEmptyMethod` method. Here, a new method syntaxtree element is created by copying the signature of the inherited method and creating a statementlist that contains a simple return statement.

Building a program generator using these metaprogramming libraries is quite similar to building an ad-hoc program generator: i.e. the generator is again written in a general-purpose programming language. The major difference with ad-hoc generators can be found in the common infrastructure that is used by the program generators, i.e. the metaprogramming library. This not only alleviates the developer from the tedious task of implementing a representation himself, it also allows for simple technical exchange of the program to be generated between multiple generators.

Macros

Macro systems are also a very well known metaprogramming facility to perform program generation. But again, many different macro systems exist and thus have a very different expressiveness and power. In general, macros are functions that are executed at compile-time and translate a part of the program in which they are used. Macros can be used for optimization by inlining of function calls but they can also serve as an implementation technique for extending the language with domain-specific constructs. A macro definition can be compared with a transformation definition, which is described in the next section. The execution of macros at compile-time is often referred

```
import openjava.mop.*;
import openjava.ptree.*;
import openjava.syntax.*;

public class AutoImplementerClass instantiates Metaclass extends OJClass
{
    public void translateDefinition() throws MOPException {
        OJMethod[] methods = getInheritedMethods();
        for (int i = 0; i < methods.length; ++i) {
            if (! methods[i].getModifiers().isAbstract()
                || methods[i].getReturnType() != OJSystem.VOID
                || hasDeclaredMethod( methods[i] ))  continue;
            addMethod( makeEmptyMethod( methods[i] ) );
        }
    }
    ....
}

    private boolean hasDeclaredMethod( OJMethod m ) {
        try {
            getDeclaredMethod( m.getName(), m.getParameterTypes() );
            return true;
        } catch ( NoSuchMemberException e ) {
            return false;
        }
    }

    private OJMethod makeEmptyMethod( OJMethod m ) throws MOPException {
        /* generates a new method without body */
        return new OJMethod( this,
            m.getModifiers().remove( OJModifier.ABSTRACT ),
            m.getReturnType(), m.getName(), m.getParameterTypes(),
            m.getExceptionTypes(),
            new StatementList( new ReturnStatement() )
            );
    }
```

Figure 2.2: Program generator for 'automatic methods' written in OpenJava

to as macro expansion. This is because macros operate in place by transforming the syntactic language construct they define into native language constructs. Macros are a powerful kind of program transformations integrated as a metaprogramming facility in a general-purpose language. C macros offer a simple textual expansion but Lisp and Scheme macros are much more powerful because they operate on the program representation rather than on strings. The R5RS Scheme [KCR98] macro facility is amongst the most advanced macro systems available in programming languages today. It offers a template-based, hygienic rewriting facility.

2.7 Transformational Generators

Transformational generators constitute a large body of generators being used today. These generators are implemented using a general program transformation system. Although many different kinds of those transformation systems exist, they always have a transformation engine at their core that executes transformation rules to transform an input program into an output program. Although program transformations may be expressed in any programming language, specialized transformation languages are more appropriate to express program transformations. This is because transformation languages provide specialized support for operations frequently needed to implement transformations. Operations such as pattern matching, querying and traversals are native to the transformation language, while they need to be implemented by hand in a general-purpose language, which is often quite a cumbersome job. Other important features such as backtracking of transformations, dependency analysis and scheduling the application order of the transformation rules, are important features often supported by the transformation system.

In general, a program in a transformation language consists of a set of transformations. Each transformation specifies a mapping of (a part of) the input program to (a part of) the output program. Two fundamentally different kinds of transformations exist: *forward* and *reverse* transformations. Forward transformations are source-driven. This means that the output program is constructed by walking over the source program and applying transformations. Reverse transformations are target-driven: the output program is a template that is filled in by querying over the source program. Both kinds of transformations are not mutually exclusive and some systems support both, such as XSLT [Tid01]. There are other important differences between transformations such as their scope and stages of the transformation process. We do not consider these differences here and refer the interested reader to a survey on transformation mechanics [vWV03].

The most simple kind of forward transformations are rewrite rules. A rewrite rule consists of a pattern that needs to be matched in the input and a pattern that is produced in the output when the rewrite rule is applied. The following rewrite rule specifies that the input pattern `double(X)` must be replaced by the output pattern `2*X`, where X is a variable in the pattern:

```
double(X) -> 2*X
```

Such a rewrite rule will, for example, transform `4 + double(4)` into `4 + 2*4`. A rewrite rule is applied by the transformation system if the input pattern of the rule can be matched in the input program. In most cases, the system will continue executing rewrite rules as long as any rewrite rule is still applicable. The rewrite rule matches a pattern in the input program's text. However, in most cases, transformation systems operate on an internal representation of the input and output program, which is most often an abstract syntax tree. The rewrite rule mechanism is the basic technique underlying forward transformation technology. The input program is gradually transformed into the output program. In each transformation step, a pattern in the input program is matched and a corresponding pattern is produced in the output program. More advanced transformation techniques such as the Scheme macro facility (which was described earlier) still follow the same idea of rewrite rules. However, they add a lot of additional power. For example, Scheme macros can use the full programming power available in the Scheme programming language to implement their transformations.

Reverse transformations are very different from forward transformations. They are based on queries over the source program to construct the output program. This kind of transformation is more adequate if the output program is rather fixed and only needs some customizations that are driven by the input program. Reverse transformations emerged in template-based generation of webpages or programs [vWV03]. For example, consider the following template (in pseudo code) to generate a webpage. The output of this (reverse) transformation is an html webpage that contains a `title` and a `content` that are obtained from the input program by launching the `getTitle()` and `getContent()` queries.

```
<html>
<head>
<title> <query> getTitle() <query> </title>
<body>
  <query> getContent() <query>
 </body>
</html>
```

Obviously, too many transformation systems exist to describe them all in detail here. Therefore we limit ourselves to some key techniques that are often used in the context of generative programming.

2.7.1 Draco

Draco [Nei80, Nei89] is an approach to domain engineering using domain-specific languages and transformation technology, designed and implemented by John Neighbors [Nei80]. The main goal is to bring the reuse in software engineering from the implementation phase to the design and analysis phase. Reuse of design and analysis is achieved by writing software in domain-specific languages. Domain-specific languages are different from general-purpose languages because they typically allow to describe a problem at a higher (domain-specific) level in which the requirements and/or design are explicit. These languages encapsulate the knowledge of a particular domain and have been carefully designed and tailored by domain-analysts. Hence, programs written in domain-specific languages explicitly describe their requirements and/or design, which would have been lost if they were directly implemented in a general-purpose programming language. Program generation is an essential part of Draco as the domain-specific program is a high-level description from which a program in a general-purpose programming language is generated.

The domain-specific languages in Draco are implemented using a (forward) transformation system. The transformations operate on the internal (parsetree) form of the program and translate it into a program in another language. This might again be a domain-specific language, meaning that the program needs to be translated further on, until it is expressed in an executable language. For this purpose, Draco makes a (conceptual) distinction between application-, model- and execution domains. Application domains encapsulate knowledge about a particular class of applications, such as spreadsheets, broadcasting, banking, ... Modeling domains are used to encapsulate knowledge about parts that can be used to implement applications, such as databases, graphics, numerics, ... And finally, execution domains are concrete programming languages such as Java, C++, Smalltalk, ... Languages in the application domain are implemented in terms of languages in the modeling domain. These languages are, in turn, implemented in execution-domain languages. This means that a program, written in a particular application-domain language, will be subsequently refined into (perhaps many) model-specific languages and eventually into a program in a general-purpose language. This setup is illustrated in figure 2.3.

The translation process in Draco uses two kinds of transformations: optimizations and refinements. Optimizations are intra-domain transformations, meaning that they rewrite a program to a program expressed in the same domain. This is often done for simplification or optimization of the program. The following transformation rule is a simplified example of an optimization rule for a mathematical language implemented in Draco. The rule is named `ADDX0` and it specifies that the addition of any term `X` with zero is the term itself. Obviously, these rules follow the rewrite rule paradigm.

```
(TRANS ADDX0 (ADD X 0) X)
```

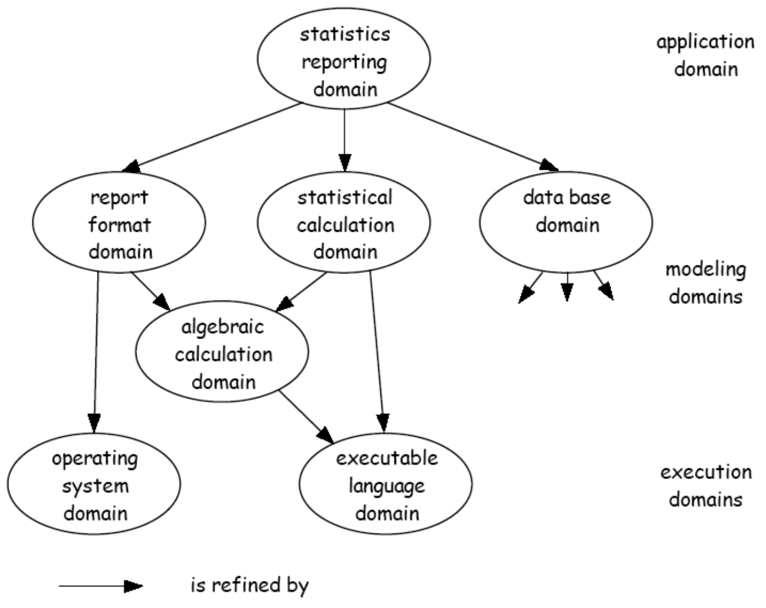
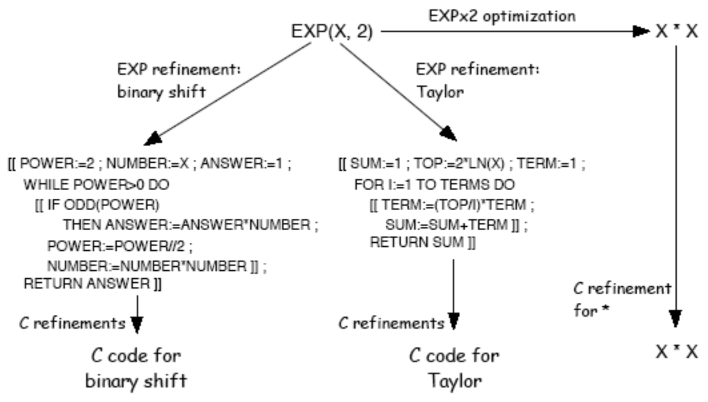


Figure 2.3: Stepwise refinement through Draco domains (from [Cza98]).

Refinements are inter-domain transformations and 'refine' a domain-specific program to an executable program. Refinements transform the internal representation of a program in a certain domain to the external or internal representation of the program in another domain. Refinements can be seen as the mapping of a domain-specific language element to its implementation. There can even be multiple refinements for the same language element. This means that there are multiple ways to transform a program to its executable implementation, especially if we also consider the application of the optimization transformations. Figure 2.4, illustrates the multiple ways in which an exponentiation expression may be refined to its implementation. It is possible however, that a particular refinement produces an implementation that conflicts with the subsequent refinement of that implementation. Therefore, refinements are equiped with conditions and assertions. The conditions of a certain refinement ensure that it is only executed if the conditions are true. The assertions are annotations that are attached to the resulting implementation and can be used by the conditions of further applicable refinements on that implementation.

The translation of a (domain-specific) program in Draco is a semi-automatic process, where the system may ask the user to suggest the next translation step. In order to prevent the system from asking too much questions during the translation process, the developer may specify a set of tactics or strategies in the system. Tactics



```
[[ POWER:=2 ; NUMBER:=X ; ANSWER:=1 ;
  WHILE POWER>0 DO
    [[ IF ODD(POWER)
          THEN ANSWER:=ANSWER*NUMBER ;
      POWER:=POWER//2 ;
      NUMBER:=NUMBER*NUMBER ]] ;
  RETURN ANSWER ]]
```

```
[[ SUM:=1 ; TOP:=2*LN(X) ; TERM:=1 ;
  FOR I:=1 TO TERMS DO
    [[ TERM:=(TOP/I)*TERM ;
      SUM:=SUM+TERM ]] ;
    RETURN SUM ]]
```

Figure 2.4: Alternative refinement paths for EXP(X,2) to a C program (from [Cza98]).

and strategies are guidelines that help the system to determine when to apply which refinement. Furthermore, domain-specific procedures can be specified whenever a set of transformations can be applied algorithmically.

2.7.2 Intentional Programming

The primary focus of the Intentional Programming system (IP) is modular language implementations [Cza98]. In IP, a language is implemented as a set of modular parts, each implementing a particular language abstraction. The modularity of these parts facilitates their reuse in other language implementations, as well as the implementation of new language abstractions in an existing language. This is in contrast with traditional language implementations (i.e. compilers) that are very hard to extend or modify. Hence, it is no mystery that the IP technology is of primary interest to domain-specific language implementers because it especially facilitates the building of domain-specific languages as a set of modular parts. We already explained how program generators are domain-specific language compilers, so it should be clear that the IP system is a program generator technology.

The IP system calls these modular language abstractions *intentions*, referring to the IP vision that a programmer should express his intentions explicitly in the code, rather than implicitly using inadequate language features. This vision is shared by designers of domain-specific languages, in which adequate language abstractions are used to reflect the domain and its operations. Implementing a domain-specific language in the IP system boils down to implementing a set of intentions. For the purpose of this dissertation, we will discuss intentions from a program generation viewpoint.

That is: each intention defines a (forward) transformation that implements the semantics of the intention's language abstraction by generating program code for it. But intentions define much more than transformations. An interesting aspect of IP is that a source program is not represented as text but as active source, that is, as a data structure with behavior at programming time. This means that besides the definition of a transformation, each intention defines how it should be visualized in the program source (e.g. as a mathematical formula, a UI spec, ...), how it should behave in the debugger, how it behaves in the version control system, etc.... Each of these functionalities is defined by a separate method on the intention module, much like methods of classes in object-oriented programming.

The system triggers the necessary functionalities by invoking the appropriate methods on the active source representation of the program. The active source is a tree of nodes where each node is an instance of a particular intention in the input program. In fact, the tree is actually a graph because there are not only links that reflect lexical relationships in the program structure, but also links that denote dependencies and other relationships between nodes. The concept of active source and other important particularities are equally important to the IP system. In the remainder of this section, we limit our discussion of IP to the program generation technology it uses. For other aspects of IP, we refer the interested reader to [Cza98, MSvW01, ADK$^+$98].

Figure 2.5 shows a part of the active source tree for the expression `x+y+z`. The full lines show the sourcetree structures and correspond to parent-child links in the tree. The dashed lines show relations and dependencies between the nodes. In this example, a use of a variable or an operator points to the corresponding declaration.

Reduction

IP refers to the program generation process as the reduction process. During reduction, the original source program is incrementally transformed to the low-level implementation. Each intention performs his part of the transformation process and transforms a small part of the source program. An intention can either transform directly to the low-level language, or it can generate code that will (partially) be transformed by other intentions.

An intention specifies how it should be reduced by means of a reduction method. In IP terminology, the program code produced by an intention's reduction method is called the *Rcode* of the intention. The system starts the reduction by invoking the reduction method on the root node of the source tree. The root node subsequently invokes the reduction method on its child nodes and uses the resulting Rcode to produce his own Rcode representation. Furthermore, during reduction, each node can also ask information from other nodes in the source graph.

As in each transformation system, the order of application of the reduction meth-

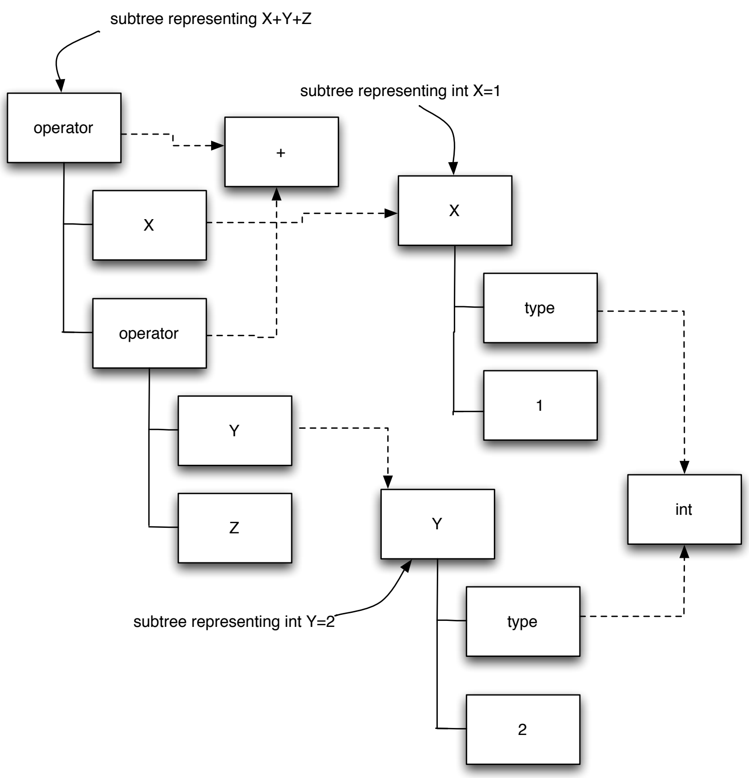

subtree representing X+Y+Z
operator
+
subtree representing int X=1
X
X
type
operator
1
Y
Y
int
Z
subtree representing int Y=2
type
2

Figure 2.5: Source tree for `x+y+z` and `int x=1`, `int y=2` (adapted from [Cza98]).

ods is often quite important. Different orderings of reductions might result in different result programs, which may or may not be correct. In general, this is because reductions change the source graph and might influence each other's result through these changes. The problem of ordering transformations is commonly referred to as the scheduling of transformations. In an open system where new transformations can be introduced in the system, it is impractical to let the developer specify the schedule for a particular set of transformations. Whenever new intentions are added, the schedule should be revised, requiring a detailed analysis of the influences between the intentions. This obviously requires detailed knowledge of the particular intentions. To overcome this, the reduction methods of all intentions in IP have to adhere to a few basic principles such that the transformation schedule can be determined by the system itself. The general idea behind the following principles is that the reduction method of each intention can assume that the entire source graph is already in its final state, except for the changes to be performed by the reduction method itself:

- Reductions cannot remove nodes or links from the source graph. Each reduction actually attaches the resulting Rcode to the source graph. The source graph grows during the reduction process until the entire program is reduced. Because reductions cannot remove information from the source graph, the reduction process will always terminate.

- A method that is executed on a node may only access neighboring nodes in the source graph and nodes that were passed as arguments of the method. Furthermore, a method may only add new links to the node it is executing on. If a method needs information from a distant node in the source graph, the neighboring nodes should have methods that forward the method invocation to their neighbors and so on until the desired node is reached. The advantage here is that the system knows which nodes use information from which other nodes during the reduction process. This means that the system can build an overview of the dependencies between the different intentions. This information is important to support the next principle.

- The answer to each question may not change during the entire reduction process. This is monitored and enforced by the system. If the information in a particular node is changed (e.g. by adding a new link to the node), all methods that were already invoked on that node are re-executed and the results are compared with the previous executions. When the results have changed, the system rolls back to an instant in the reduction process where the methods were not yet invoked and tries invoking the methods in a different order. This is possible because of the following principle.

- Method invocations can occur asynchronously. The system can then decide in what order these methods are actually executed. The more method invocations occur asynchronously, the more possible orderings the system can try.

As a result, the system can try to find a correct application order for the reductions, such that a particular reduction does not invalidate the results of a previously executed reduction. This is opposed to having a fixed transformation order for a set of intentions, which complicates the extensibility and composability of intentions. In most of the cases, the IP system will be able to schedule the reductions in a correct order. The reduction process can only fail to find a schedule if there are reductions that change the same intentions in incompatible ways. This might happen if an extension library contains reductions that change intentions in the language being extended.

2.7.3 XSLT

The XSLT language [Tid01] is commonly used to transform XML documents into something else. The result of the transformation may be another XML document, an HTML or even a PDF document. Although it was not intended as a program transformation system but as a document transformation system, XSLT can be used to transform programs. This is because an abstract syntax tree of a program can also be represented as an XML document. Obviously, representing the parsetree of the source program as an XML document is a prerequisite.

XSLT has the interesting feature of supporting both forward and reverse transformations. An XSLT program, or so-called stylesheet, contains a number of forward transformations that are applied to the source XML document to produce a target XML document. Like any forward transformation, each XSLT transformation consists of a pattern that needs to be matched in the source document and the corresponding result pattern in the target document. In XSLT, these transformations are called 'templates', referring to the template result pattern. For example, the following template (adapted from [Tid01]) transforms occurrences of an XML tag `<greeting>` to an HTML document displaying the (textual) contents inside the `<greeting>` and `</greeting>` tags. The source pattern is described inside the `<xsl:template match="greeting">` tag, which says that this XSLT template transforms occurrences of the `<greeting>` tag. Inside the result pattern, we use the `<xsl:value-of select="."/>` tag to retrieve the (textual) contents between the `<greeting>` and `</greeting>` tags.

```
<xsl:template match="greeting">
  <html>
    <body>
       <p>
```

```
        <xsl:value-of select="."/>
      </p>
    </body>
  </html>
</xls:template>
```

The application of the templates on an XML document automatically occurs in a recursive fashion, until a template is found that transforms a particular subtree of the XML document. The application of any other templates for the transformation of that subtree is entirely determined by the template that matches on the root of that subtree. For example, in the example above, no more templates will be executed on the subtree beneath the `<greeting>` and `</greeting>` tags. This is because XSLT requires that an explicit control flow is defined on the application of the templates. The standard control flow is one that recursively descends the xml document and tries to match any template. The standard control flow is overridden if a user-defined template matches a particular tag. The application of templates is expressed by explicitly calling a template on (a part of) the subtree (e.g. `<xls:apply-templates select="greeting">`). To further control the application of templates, XSLT provides the developer with control flow constructs to implement iterations (`<xsl:for-each>`) and branches (`<xsl:if>`,`<xsl:choose>`). In general, we conclude by saying that XSLT transformations and the transformation application control flow are tangled.

Besides the use of forward transformations, XSLT provides support for querying the source document through XPath expressions. The result of these queries are used inside the result patterns of the templates. We are not going to discuss the details of the XPath query language here, but merely illustrate its usage. In the previous example, we already used it to retrieve the contents of the current node under transformation (i.e. the `"."` in the `<xsl:value-of select="."/>` construct). Using XPath, we can retrieve information from anywhere in the source XML document by expressing a path over the tree that starts at the current node under transformation. This allows us to implement reverse transformations because we can 'fill in' a particular result pattern with information retrieved from the source document. For example, the following template transforms the root node of the source tree into an html document and retrieves its information from the source tree using XPath. It accomplishes the same transformation as our previous example but it uses XPath to query the subtree of the rootnode (which is the current node under transformation) to retrieve the `<greeting>` tag.

```
<xsl:template match="/">
  <html>
```

```
    <body>
       <p>
         <xsl:value-of select="greeting"/>
       </p>
    </body>
  </html>
</xls:template>
```

2.8 Compositional Program Generation

Compositional program generators produce an output program by composing several smaller program building blocks together. The building blocks are programming abstractions such as classes, functions, components, templates, aspects, hyperslices, ... The idea is that each building block implements a particular feature and can be composed with the other building blocks through a composition technique. It is also common that a set of composition rules and constraints govern dependencies between the separate building blocks such that the generator always produces a correctly working system. The use of compositional program generators depends on whether or not we can implement the required features in separate program parts and recompose them to generate an output program. A compositional generator We describe the most important composition program generation techniques below.

2.8.1 GenVoca Generators

GenVoca [BST$^+$94b] is a design methodology for creating software product-lines [BJMvH02]. GenVoca has now been generalized into the AHEAD model[BSR03] but we restrict ourselves to the discussion of the original GenVoca model because its generalization is not important with respect to our discussion.

In the GenVoca model, a software application is generated through the composition of layers of abstraction. Each layer implements a particular feature and consists of abstractions native to the programming language (e.g. classes, methods, functions, templates, mixins,...). Stacking layers onto each other yields a complete application containing the features implemented by the respective layers. This is because each layer 'refines' the layer above it by composing its internal abstractions with the already existing abstractions in the layers above or by adding new ones. Several different implementation technologies have been used to implement the GenVoca model. The most prominent and well-know examples are through C++ templates [Cza98] and Java mixin-layers [SB98]. In both these implementation techniques, the object-oriented inheritance is used to compose the different layers.

In order to obtain a particular application, we need to describe the desired composition of layers in a GenVoca equation. Conceptually, in these equations, programs are values and refinement-layers are functions. These functions take a program as input and produce a program refined with the particular feature (implemented by the layer) as output. For example, consider the following equations:

```
application1 = f(g(x))
application2 = h(i(x))
```

In this example, we define two applications. `application1` is the program x, extended with the features `f` and `g` and `application2` is program x, extended with features `h` and `i`.

In the GenVoca model, the generator's implementation is based on language features available (or integrated) in the general-purpose language. For the purpose of implementing GenVoca layers and generators in Java, the language was extended with mixins and mixin layers. In short, a mixin in Java is a class without a static superclass. This means that the superclass of this mixin class is not specified at the definition of the class. Instead, when the mixin class is used, it must be supplied with a superclass, which can again be a mixin class. As such, we can use the same mixin class to extend the behavior of many other classes. To use a mixin class, we define a new class that is the composition of the mixin with its superclass. This is done through the typedef construct. The following example illustrates the composition of a mixin M with a class C into the new class N.

```
typedef N M < C >
```

A mixin-layer is a mixin that contains other mixins and classes. It is used to group mixins together. Clearly, mixin layers are used to implement GenVoca layers and mixins and normal classes are the basic abstractions inside each layer. The composition of genvoca layers is thus implemented as the composition of mixin layers. Inheritance between mixin-layers is defined in terms of inheritance of its parts. In figure 2.6, the inheritance hierarchy for a composition of layers `x`, `f` and `g` is shown. Layer `x` is the core of the application and is refined by layers `f` and `g`. Therefore, layer `x` is implemented using normal classes, while the other layers consist of mixins. The resulting application is the composition of these layers or, technically, the set of most specialized subclasses of each inheritance chain.

Of course, not all layers can be stacked onto each other and some compositions might not even result in a working system. These problems are respectively solved by a type-checker and a design-rule checker [BG97]. The type-checking is based on the fact that layers are grouped in realms and each layer can only accept layers of a particular realm as input. This means that the result of certain functions cannot

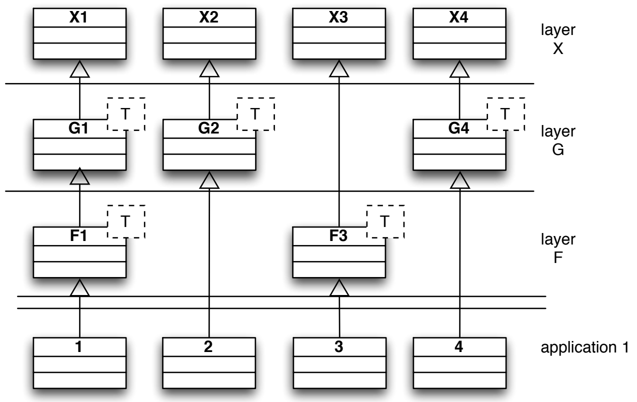


Figure 2.6: The inheritance hierarchy implementing the `f(g(x))` GenVoca equation.

be used as a parameter of other functions. A realm can thus be seen as a typing mechanism for layers and the arguments and return value of a function in a GenVoca equation are statically typed. The correctness of the equations is then checked by a type-checker. More complex, *semantic* design-level dependencies are expressed by adding applicability constraints to the layers. These constraints can describe the incompatibility of features or they can enforce a certain order on the stacking of the layers, etc. . . .

2.8.2 Aspect-oriented Programming

Aspect-oriented programming [KLM$^+$97] is a novel programming technique to modularize the implementation of so-called *crosscutting concerns*. Crosscutting concerns are parts of a program that cannot or cannot be easily modularized using contemporary programming languages. Often cited examples of such crosscutting concerns are synchronisation, error-handling, persistence, security, etc. . . . In an aspect-oriented programming language, a novel modularization is offered in which the implementation of such a crosscutting concern can be syntactically modularized. Such a module is commonly referred to as an *aspect* and consists of two main parts: the aspect functionality and a *pointcut* or *crosscut* definition. This crosscut definition specifies where or when the aspect's functionality needs to be invoked. This is where aspects differ from traditional language modules: the aspect itself specifies where or when it needs to be invoked. The first aspect languages were domain-specific languages to describe, for example, synchronisation (COOL [Lop97]) or loop-fusion (RG [MKL97]). Afterwards, more general-purpose aspect languages were developed, such as the well-known AspectJ aspect language [KHH$^+$01]. AspectJ is an extension to the Java programming language with aspects.

In aspect-oriented programming, the program that actually executed is produced by an aspect weaver. An aspect weaver accepts the definition of a set of aspects and the so-called base program and *weaves* the aspect implementation (its functionality) into the base program at the appropriate locations defined by the crosscut. Such an aspect weaver can be seen as a program generator that modifies the original program and composes it with the aspect implementation. Consequently, aspect weavers can be seen as compositional program generators.

2.8.3 Subject-oriented Programming and Multidimensional Separation of Concerns

In Subject-Oriented Programming (SOP) [OKK$^+$96, HO93], an application is built through the composition of subjects. Each subject is a collection of program parts and the composition merges appropriate parts together to build the resulting program. A subject may be a complete application by itself or it may be an incomplete fragment that needs to be composed with other subjects. As in all composition-based techniques, the idea is that each subject implements a separate feature of the entire program and the composition of subjects integrates their corresponding features in the output program. The composition of subjects is governed by composition rules that establish correspondence between entities in different subjects. The corresponding entities are then merged together in a specific way, which is also indicated by the appropriate composition rule. There are many different composition rules and they are described separately from the subjects, which is in contrast with the mixin-implementation technique for GenVoca, where one kind of object-oriented composition technique is used to compose the different parts.

Because of this diversity in composition rules, the composition of subjects is quite flexible and facilitates the reuse of subjects across different applications. As we already mentioned, a subject is a syntactically correct program, implementing a particular feature which may or may not be an already complete program. Since the SOP technique has been applied in practice in the context of class-based object-oriented programming, this means that subjects consist of classes, containing methods and instance variables. The composition rules are described in separate files and establish a composition between these three kinds of parts in each of the participating subjects. The rules are divided in correspondence and combination rules. The correspondence rules establish a correspondence between the different parts that need to be composed and the combination rules determine how the corresponding parts should be composed. The most basic and default correspondence rule establishes a correspondence between elements of the different subjects that have the same name. Exceptions to this rule can be manually described. A combination rule for different parts either *joins* the parts together or one of the parts *replaces* the other. Furthermore, ordering constraints can be specified to control how method bodies are combined and a function can be specified to compute the return value of a composed method.

The subject-oriented programming technique has now evolved into multidimensional separation of concerns (MDSOC) [OT99], of which it is now a part. MDSOC is also mentioned as an aspect-oriented software-development approach and is a generalization of subject-oriented programming to all phases of the software development lifecycle. It also includes a number of improvements and extensions to the composition rules, which we do not describe here as the general concept remains the same.

Chapter 3

Analysis of Program Generator Composition

In the previous chapter, we provided an overview and a classification of existing generative programming techniques. The primary goal of these techniques is the development of program generators that generate black-box programs. Although this black-box property is desirable to hide implementation details and prevent broken functionality of the generated programs, it limits the reusability of the program generators. In this chapter, we explain how black-box generated programs prevent a composition of generators such that their respectively generated programs are integrated. More importantly, we identify a set of requirements for a composition technique for program generators that combines the advantages of black-box generated programs with the ability to integrate the generated programs of different program generators.

3.1 Introduction

A program generator is generally considered in isolation. Using contemporary generative techniques, a program generator produces a program that is implemented in one or more encapsulated modules with a well-defined interface. The application developer can use the generated program by accessing this interface. The internal implementation of this generated program is hidden and considered as a *black-box* for the application developer who applies the generator. Consequently, the application developer cannot and should not change any internal implementation details of the generated program. He can only use the mechanisms available in the programming language (such as parameterization and inheritance) to influence or adapt the behavior of the generated program. Although a mechanism such as inheritance already provides some powerful adaptation possibilities, it also requires more detailed knowledge on the internal implementation to achieve a correct adaptation. This conflicts with the black-box property of generated programs. Nevertheless, the black-box property is desirable because it hides the application developer from the complexity of the generated code. In essence, most of the internal implementation details of a generated program are completely determined by the program generator. However, using multiple program generators in the development of a single application will inevitably require a composition of the generated programs. The black-box property of the generated programs implies that we can only compose such generated programs by means of glue-code. However, such glue-code composition of black-box generated program modules is not sufficient. First of all, many frequently implemented parts of programs *are* not or *can* not be implemented using encapsulated modules. Examples of such program parts are collaborations between multiple modules (e.g. the implementation of publish-subscribe dependencies) or conceptual modules that are spread across different implementation modules (e.g. synchronisation code). This is also illustrated by the advent of aspect-oriented software development [KLM+97, Fil02] that refers to these program parts as *crosscutting code*. Although there is a clear need for generators of crosscutting code, composition of the generated crosscutting code with other generated programs is problematic. This is because crosscutting code requires to be *integrated* into a (generated) program. The black-box property of generated programs hides all internal implementation details which severely restricts the possible integrations. In essence, we can only express an integration with the public interface (methods) of a generated program. More invasive integrations require more detailed knowledge on the internal implementation of the generated programs. Furthermore, even if we can access the internal implementation and specify a more invasive integration of the crosscutting code in the generated program, this integration can still lead to undesired interactions and break the functionality of the generated programs. Such undesired interactions (or interferences) are even more apparent when multiple crosscutting code parts are integrated in a (generated) program. On the other hand, interactions are also frequently intended or required in particular integrations. This means that in some integrations, such interactions need to be prevented, while in other integrations, the interaction is part of the integration itself.

As we will illustrate later on, even if we do not consider crosscutting code, particular compositions of generated programs require interactions that are hard or even impossible to achieve in a given programming language by means of glue-code, inheritance or parameterization . Instead, such interactions also require an invasive integration of the generated programs. To achieve an invasive integration of generated programs, we propose a particular composition of program generators that we will refer to as *integrative composition*. In an integrative composition of two generators, each generator can perform an invasive modification to the generated program of the other generator. This means that parts of the implementation of a generated program are inserted in, or *integrated with*, another generated program. The result is a merge of the generated programs that combines their individual structures and functionalities. Clearly, this requires a composition technique that breaks the encapsulation boundaries of the generated programs. Furthermore, the composition mechanism needs to prevent undesired interactions (interferences) but needs to be sufficiently flexible to allow the intended interactions between the generated programs.

In the following section, we describe modular program generators and why integrative composition is important in their implementation. Next, we describe how functional composition is inadequate to support such integrative composition and finally, in section 3.5, we discuss the requirements for an adequate integrative composition technique. The actual generative technique that is designed to allow integrative compositions is described in the following chapter.

3.2 Modular Program Generators

It is considered good practice to implement individual features of a software program in separate modules and use a composition technique to recompose the separate modules into a working program. Each of the separate modules themselves is again a program that can be decomposed into separate modules. This principle of separation of concerns and hierarchical decomposition [Dij76] is a driving force in software engineering and programming language technology. The more we separate the implementation of individual features of a program into separate modules, the more we can reuse these individual modules in the implementation of other programs, and thus, reuse the feature they implement. The same principle holds for the development of program generators. By modularizing the implementation of a program generator, we can reuse the individual modules in the implementation of other program generators. In this section, we first motivate why modular generators are best achieved through a modular composition of program generators and illustrate this motivation by means of a set of small example generators. We also describe the two different kinds of composition that are needed to build meaningful compositions of these generators. In particular, we discuss the difficulties involved with integrative composition.

3.2.1 Motivations

Separation of Generated Concerns

In applying the principle of separation of concerns [Dij76] to the implementation of a program generator, it is best to modularize according to the concerns that need to be generated. This means that the generation of each concern of the output program needs to be implemented in a separate module of the program generator. For example, a generator that produces a traversable tree implementation can consist of a module that implements the tree implementation structure and a module that produces the traversal methods that need to be integrated in the tree implementation. This modularity is desirable because the implementation of each module can concentrate on the generation of a single concern. Consequently, such a modularity structure is a natural way of structuring the implementation of a program generator. Because the individual modules implement the *generation* of a particular concern, they can be reused in the implementation of other generators that also require the generation of that concern. It also means that a crosscutting concern in the generated program corresponds to a modularized concern in the program generator.

Separation of Generation and Integration

To maximize reusability of the individual modules, their coupling needs to be reduced to a minimum. It means that each module must make as few assumptions as possible about the other modules. In program generators, this does not only mean that each module may not be concerned with *how* another part of the program is generated, but also that each module may not be concerned with *what* the generated programs produced by the other modules look like. Each module may only be concerned with the generation of its particular part of the program and may not be concerned about how it must compose or integrate its part of the program with the other generated program parts. The integration itself must be a property that is determined by the composition of the individual modules.

For example, the module that generates the traversal code must only concentrate on the generation of the traversal methods. These methods need to be integrated in the tree implementation. However, to produce these methods, the traversal generator must know how the child nodes of a node can be retrieved. Furthermore, to produce a complete implementation for a traversable tree, the methods need to be included in the implementation of the classes that implement nodes and leaves. The module that produces the traversal code is strongly coupled to the module that produces the tree implementation if it can only produce traversal code for that particular tree implementation. For example, this occurs if the generated traversal code assumes the existence of a method with a particular name and signature in the tree implementation that retrieves the child nodes. To achieve a minimal coupling, the modules need to be parameterized with information required from other modules. In the example of the traversable tree, the module that produces the traversal code needs to be parameterized with the method (included in the tree implementation) that can be invoked to obtain the child nodes. Another parameter is the exact location where the methods need to be integrated in the tree implementation. This means that the specification of the composition or integration of the generated program parts (produced by each individual module) needs to be specified external to the implementation of the modules.

A separately specified composition or integration of the generated program parts allows that the generated program part of a module is not specifically implemented for integration with a specific other program part. In this way, each generator module makes as few assumptions as possible about the other modules and their generated program parts, which enhances the reusability of each module.

Composition of Program Generators

The composition technique to compose the generator modules, mentioned in the previous sections, needs to consider these individual modules as individual program generators. This is because each module can only produce its own program part and must be parameterized with information that is needed from other modules to generate its own part of the program. Consequently, a module behaves as a stand-alone program generator. At least, the composition technique will need to consider the modules as individual program generators. Such a composition technique for individual generators will not only allow to build modular generators but will also allow to compose existing generators. Another advantage of building modular generators as a composition of other generators is that the generation of a single concern (implemented by a single generator) can also be useful in manually implemented programs. In other words, a generator that produces the implementation of a crosscutting concern can also be useful when considered in isolation. A *modular composition technique* for program generators does not only allow us to build compositions of existing generators but also allows us to build *modular generators* of which the individual modules are program generators themselves. More specifically, a modular composition technique for program generators that addresses the integration of the generated programs will allow developers to compose the functionality generated by each program generator into a single generated program.

3.2.2 Examples

Building a program generator as a modular composition of generators that each produce an individual concern of the generated program, is a natural way of structuring a program generator. To clarify this, we introduce two example program generators without delving into their particular implementation details. A first example introduces a tree generator which is composed with traversal and balancing generators. A second example focuses on the composition of a graph generator and an observer-observable generator. Each of the examples illustrates how separation of generated concerns is useful to structure the modularization of a program generator. We will use these examples throughout this chapter to illustrate the problems associated with the modularization and the required composition. We also implement them in the following chapters.

Tree-implementation generator

The tree generator produces an object-oriented implementation of a tree structure. This generator accepts a specification of a tree implementation consisting of names of nodes and leaves, as well as a specification of restrictions on the structure of the possible trees that can be built with it. This specification restricts which kind of nodes may be a child of a specific kind of node and how many children this particular kind of node may or must have. In figure 3.1, an example specification for a tree implementation is shown. The syntax of the specification is not important but it shows that in a tree implementation specification, a tree is modeled by the names of the nodes and the leaves. Each node (e.g. `NodeA` and `NodeB`) has a list of possible children associated with it. In the example, `NodeA` can only have children of the kind of `LeafA` and `NodeB`. Furthermore, each node also specifies its arity (the number of children it (can) have) and whether it is a fixed or maximum number. For example, `NodeB` is restricted to a maximum of 4 children.

```
Node NodeA
  children LeafA,NodeB
  multiplicity fixed 2
Node NodeB
  children LeafB,NodeA
  multiplicity maximum 4
Leaf LeafA
Leaf LeafB
```

Figure 3.1: An example tree specification.

The generated implementation for a tree consists of a set of classes, corresponding to the specified nodes and leaves. The implementation that is generated for the tree specification of figure 3.1 is illustrated as a UML model in figure 3.2. Some common behavior and state for nodes and leaves are implemented in the classes `SuperLeaf` and `SuperNode` respectively. Each (concrete) node class implements the necessary methods to attach nodes, remove nodes, etc.... Furthermore, all generated methods implement the necessary checks to prevent any incorrect tree structures. These checks prevent the creation of trees that do not correspond to the restrictions specified in the input specification and prevent the creation of cycles in the tree. For example, the generated implementation of the `child:at:` method on `NodeB` (that adds a child node to a node of kind `NodeB`) is shown in figure 3.3. Furthermore, a value can be contained in each node and leaf and can be accessed through generated set and retrieve methods. Summarized, the generated implementation contains all structures and behavior that is necessary to build a correct tree and is tailored towards the requested features.

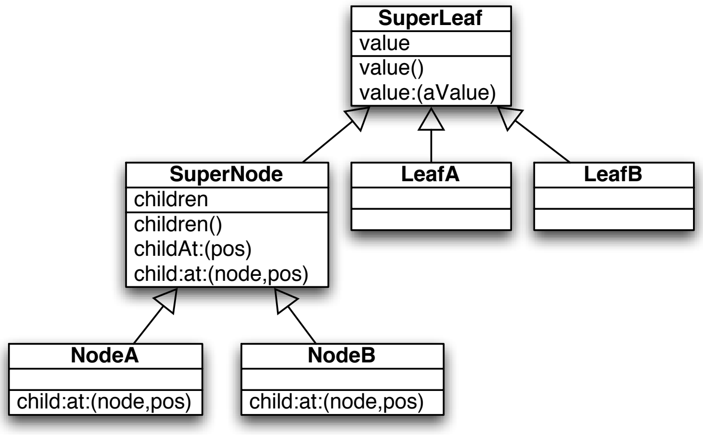


Figure 3.2: An example tree implementation.

```
NodeB >> child: aChild at: aPosition

((aChild isKindOf: LeafB) or: [aChild isKindOf: NodeA])
     ifTrue:[self error:'Creating incorrect tree']
     ifFalse:[children at: aPosition put: aChild]
```

Figure 3.3: Generated implementationf for the `NodeB` >> `child:at:`

Contemporary implementation techniques such as object-oriented frameworks and library components cannot be used to implement the variability and tailorability with the same efficient implementation as the one produced by the tree generator. On the one hand, a (static) library cannot implement all possible tree implementations that can be specified. A tree generator, on the other hand, can produce an entire family of possible tree implementations. The tree generator automates the process of writing the different implementations manually. Furthermore, an implementation of the tree as a specialization of a generic object-oriented framework (or with customizable library components) has to rely on object-oriented inheritance or parameterization to implement the possible variabilities of each tree implementation. Besides the names of nodes and leaves, there is a variability in the generated code of the `child:at:` method that ensures that a created tree adheres to the restrictions in the specification w.r.t.

the possible child nodes. This generated code is different depending on the chosen restrictions. Using contemporary programming languages, the implementation of these variabilities is best done using parameterization. This means however, that at each instantiation of a node or leaf, the correct arguments (e.g. names of possible kinds of nodes and leaves) must be supplied. This can again be solved using the template method or abstract factory patterns [GHJV95]. However, the developer also needs to ensure the consistency of these arguments with the rest of the implementation (i.e. the given names of nodes and leaves must exist). This could also be solved by some additional code in the framework or library component that checks this. However, all this additional code further complicate the generated program and reduces the performance. Therefore it is best to generate the implementation. Furthermore, with a generator, the desired specifications of the tree implementation are described in a domain-specific language (specific to the domain of trees) instead of in the (object-oriented) solution domain. As a conclusion, the generation of a tree implementation is particularly useful, compared to more traditional implementations.

Reuse of the Tree Generator

It would be useful if we could reuse the basic tree-implementation generator in the implementation of other, more advanced, tree-implementation generators. For example, we could implement a parsetree generator, a traversable tree generator, a balanced tree generator and a traversable, balanced tree generator. All these examples would benefit from reusing the implementation of the tree-implementation generator. It means that we would like to compose this basic tree-implementation generator with other generators that produce an implementation for these more advanced features. The result of this composition is an advanced tree-implementation generator. One can also consider this composition as an extension to the tree-implementation generator to produce more advanced tree implementations (i.e. traversable, balanced, ... trees). The following paragraphs describe each advanced feature and its integration in the generated tree implementation in more detail and a last paragraph considers an evolution of the original tree-implementation generator.

Parser and parsetree generator Given a grammar description, this generator produces a parser and all structures required to build a parsetree. Building a generator for a parser and its produced parsetree structures is useful because it is impossible to implement all possible parsers and their corresponding parsetrees in a library with the same efficient implementation. Furthermore, the generation of parsers and parsetrees has already shown to be a useful application for program generators [LMB, JCC].

In the implementation of this parsetree generator, we want to reuse the existing tree generator. This alleviates the implementation of the parsetree generator because we can rely on the tree generator to produce a correct tree structure. As such, this generator should be a composition of the new parser generator and the existing tree generator. It accepts a specification of a grammar and annotations that specify the abstract syntaxtree structure and it produces a parser class and classes that implement the parsetree.

Traversable tree generator The tree implementation produced by the original tree generator implements accessor methods for the children of a node, which makes it possible to traverse a tree. However, applications that use a tree might require a particular kind of tree traversals. Traversals can be either in-order, pre-order or post-order. Furthermore, some of the nodes and leaves might not be required for a specific traversal, but they still need to be traversed in order to reach the children of those nodes.

Traversal code in object-oriented style is an example of crosscutting code. Although a traversal can be implemented without crosscutting code (i.e. in a functional style), the crosscutting implementation is more appropriate because the crosscutting traversal code can access the internal state of each object it traverses without the need for public accessor methods. Furthermore, a crosscutting implementation of traversals allows the client code to invoke the traversal in an appropriate object-oriented message sending style. The implementation of the traversal program requires a lot of repetitive code in each of the tree's implementation classes. As a consequence, it is most useful to generate its implementation. Even if particular design patterns (such as the visitor design pattern [GHJV95]) are adopted, a lot of code is still repetitive.

To implement a traversable tree generator we would again want to reuse the existing tree generator. This tree generator can produce the necessary tree implementation which is then integrated with the generated traversal code, produced by the new traversal generator. The traversable tree generator accepts a tree and traversal specifications and produces a traversable tree implementation.

Balanced tree generator The tree implementation that is generated by our tree generator does not contain any code to balance the tree structure. In the case of a parsetree, balancing is not required. However, we can think of many applications of tree structures in which a balancing operation would be useful. Hence, we want a generator for a balanced tree structure. Once again, instead of implementing this generator from scratch, it is useful to reuse the tree generator in the implementation of the balanced tree generator. As such, we need a composition of the tree generator with a generator for a balancing algorithm. The resulting generator accepts a tree specification and an ordering condition on the tree and produces a balanced tree implementation.

Traversable, balanced tree generator Now that we have a tree generator, a balancing generator and a traversable generator, we should be able to combine them to build a tree that is both balanced and traversable.

Evolution in the tree generator Consider that the original tree generator needs to be changed such that some new features (such as nodes with parent links, sorted trees,...) can be incorporated in the generated tree implementation. Another possible evolution is that we replace the existing tree generator with a new generator that produces a slightly different implementation. Obviously, we would want to reuse the balancing and traversal generators in combination with the new or changed tree generator. In other words, we do not want changes to propagate to the other generators. This is possible if the traversal and balancing generators are sufficiently generic and modular and most importantly, loosely coupled with the tree generator.

The tree generator example, and its traversal and balancing constituents in particular, illustrate the need for stand-alone, crosscutting code generators. The traversal and balancing generators each generate a single concern of a traversable, balanced tree implementation. Both the balancing code and traversal code are required to crosscut the tree implementation. It is also desirable to implement a traversal generator once and be able to use it on both manually implemented and generated tree implementations. In essence, the separate implementation of the traversal generator allows reuse of this generator on many different generated and manually implemented tree implementations. Besides the fact that the generated traversal and balancing implementations are required to crosscut, there are also some dependencies between the generated code of the different generators, such as methods or variables generated by one generator that need to be referenced, or used, by the code generated by another generator. We will come back to these issues later on.

Constraint-Network generator

In a second example, we build a generator that produces an implementation for a constraint network. The generated implementation will consist of a graph structure where the nodes represent values and the edges represent constraints. Whenever we change the value of a particular node, the consistency of the network is verified by checking the appropriate constraints. In fact, the constraints can propagate a change in one node to a change of the value in another node. The generation of this network involves the generation of a graph structure and the generation of a constraint solver. For the purpose of this example, we will focus on the graph generator and the generation of a particular part of the constraint solver that can be implemented as an observer-observable collaboration.

Graph-implementation generator Our graph generator is similar to a our tree generator. Given a description, it generates an object-oriented implementation for a graph structure. This implementation provides the necessary classes for each kind of node and edge, with graph manipulation methods implemented in them. The description also specifies particular features of the graph such as directed or undirected edges and if nodes can be connected by multiple edges, etc.... Generating a graph implementation has the same advantages as the generation of a tree structure.

Observer generator A generator for an observer-observable collaboration produces an object-oriented framework together with some crosscutting code that needs to be woven into an application. The observer framework implements a collaboration between several objects such that an object (the observable) can notify other objects (observers) whenever something interesting has changed about the observable. The sending of these notifications (by a message send) requires crosscutting code. The idea is that the observable is independent of its observers and that the observers can change dynamically. For that purpose, the observable merely invokes a message each time the observers need to be notified. This message is implemented by the observer framework and notifies all observers who can react appropriately. More information about the observer-observable collaboration can be found in [GHJV95]. A possible variation in functionality for this collaboration is that observers can be allowed to subscribe only once, as opposed to receiving multiple notifications of a single event. The generated code will thus be different if multiple subscriptions are allowed or not. Another possible variation is located in how the observer is linked to its observable. The observer can pass itself on via the notification message of the observers keep a reference to the observable.

Although the observer generator can best be implemented as a crosscutting code generator, the integration of the generated graph implementation with the observer implementation requires more than a cross-cutting of the observer code with the graph code. Figure 3.4 shows the generated graph implementation and the generated observer-observable implementation separately. The integration of these implementations that is required to implement the constraint-network generator must achieve that the `neighbours` of a node are in fact the `observers` of the observer. This integration does thus not only require a structural integration of both implementations but also requires a semantic interaction. The semantic interaction consists of a unification of the `neighbours` and `observers`.

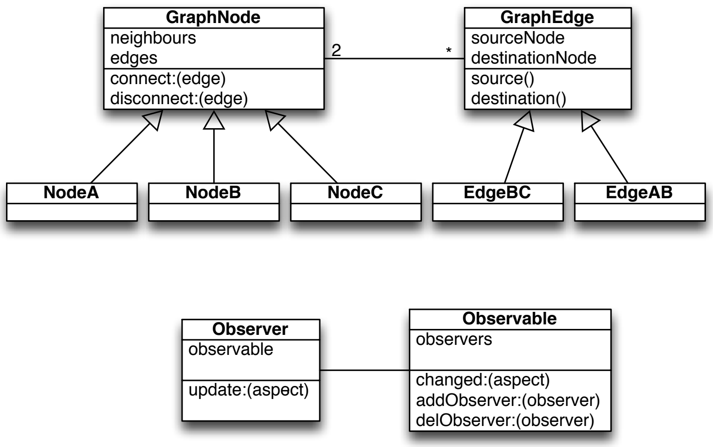


Figure 3.4: Generated graph implementation and the generated observer-observable collaboration.

3.3 Translation and Integrative Composition

Now that we have described which concerns to separate and thus how the modularity of program generators is best structured, we need a composition technique to compose the individual program generators. We can identify two kinds of composition: *translation composition* and *integrative composition*. Each of these two kinds of compositions has a totally different purpose.

The purpose of a *translation composition* is that the generated program of one generator is translated into another language by the other generator. It means that the generated program of the first generator is the input specification for the second generator. The actual translation is performed by the second generator in the composition. A translation composition is useful because the first generator reuses the generation capabilities of the second generator. It also allows us to build higher abstraction levels than those offered by the input language of a particular generator. An example of a translation composition can be found in the modular implementation of the `Parser and parsetree` generator, which is illustrated in figure 3.5. The `Parser and parsetree` generator is written as a composition of a `Parser` generator, a `Tree` generator and the `Spec` generator. The `Spec` generator accepts a grammar description where the production rules are augmented with specific annotations to permit a correct construction of a parsetree. The `Spec` generator translates this description into an EBNF description which contains production rules annotated with actions

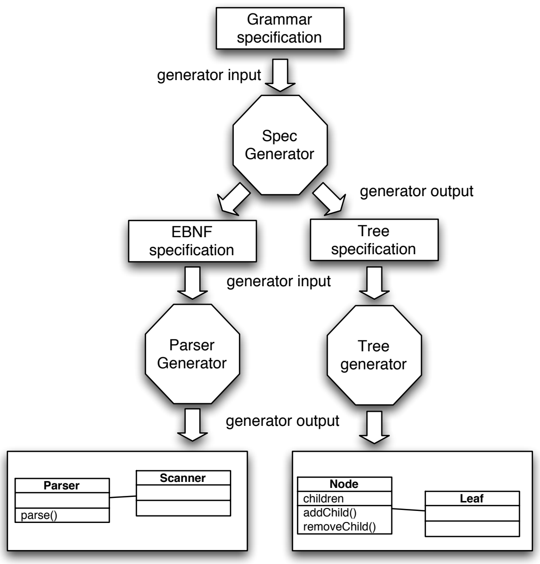


Figure 3.5: A translation composition to build the 'parser and parsetree' generator.

to build the parsetree. At the same time, the `Spec` generator produces a description of the parsetree in the tree description language which is accepted by the `Tree` generator. The `Tree` generator then further translates this tree specification into an object-oriented tree implementation. Similarly, the EBNF description is handed to the parser generator which produces a scanner and parser implementation. In the final output program, the parser and parsetree implementation communicate only through their public interfaces.

The purpose of an *integrative composition* is the *integration* of the generated programs produced by the generators. This kind of composition is required because in a modular generator, the individual concerns of the generated program are generated by individual program generators. These separately generated programs must thus be assembled (or integrated) to produce the complete generated program. In the examples we introduced above, we frequently require such an integration of generated programs. A first example of integrative composition can be found in the modular implementation of the traversable balanced tree generator, illustrated in figure 3.6. The generator is built as an integrative composition of the tree generator, the balancing generator and the traversal generator. These last two generators produce crosscutting code (variables and/or methods) that needs to be integrated in the generated program produced by the tree generator (which consists of classes). Another example of inte-

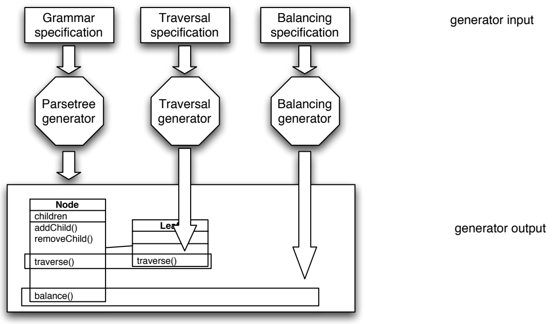


Figure 3.6: An integrative composition of a traversal generator and a tree generator.

grative composition can be found in the composition of the graph generator and the observer generator to build the constraint-network generator. The generated classes that implement the 'observables' need to be integrated with the generated classes that implement the graph nodes.

3.3.1 Requirements and Definition

In any composition, there are some basic requirements with respect to the input and output languages of a generator. For translation and integrative composition, these are respectively:

- In the case of translation composition, a generator `A` can be composed with another generator `B` if the output language of generator `A` is the same as the input language of generator `B`. Evidently, if generator `B` does not understand what generator `A` is producing, the translation composition cannot work.
- For an integrative composition, a generator `A` can be composed with a generator `B` if their output languages are the same, or can be considered as subsets of the same (integrated) language.

For example, it is clear that a Smalltalk method cannot be integrated in a Java class, or that a graph description cannot be integrated with an object-oriented program. On the other hand, the integration of an SQL program with a Java program is possible when we consider that 'Java with embedded SQL' is the common output language of both generators. While SQL and Java are different languages, we can imagine an integrative composition of generators that produce SQL and Java respectively. The result is a program written in the language 'Java with embedded SQL'.

However, this does raise some specific integration issues with respect to the embedding of SQL in Java. These issues are not considered in this dissertation. In other words, we assume that an integrated language, such as 'Java with Embedded SQL' is entirely defined.

We summarize the difference between translation composition and integrative composition in their definitions:

Translation Composition A translation composition of two generators `A` and `B` is a composition where the output program of generator `A` is the input program of generator `B`.

Integrative Composition An integrative composition of two generators `A` and `B` is a composition where the output programs of both generators are integrated into a single output program.

3.3.2 Invasive Integrative Composition

An integrative composition involves the invasive modification of the generated programs and thus requires that each generator breaks the black-box property of the other generator's generated program. This integrative composition requires knowledge about how and where to modify this generated program without breaking its functionality. In the following subsections, we explain the impact of invasive integrative composition and we also make a distinction between a structural and a behavioral integrative composition.

Integrative Composition Conflicts with Black-box Generated Programs

There is a distinction between translation of a program and modification of a program. Translation of a program changes the program's language but does not change its meaning (or behavior). Modification of a program changes its behavior but does not change the implementation language. Modification of a program requires detailed knowledge about the program in order to modify it correctly, while translation only requires knowledge about the language in which it is expressed. This distinction is important because it imposes particular requirements on a composition technique for integrative composition. Obviously, in an integrative composition, the generated program is modified because it (invasively) integrates with another generated program.

A translation composition only requires that the generated program of the first generator is written in the input language of the second generator, while an integrative composition requires that one or both generators have detailed knowledge about the generated programs of both generators. This implies that integrative composition conflicts with the black-box property of generated programs. The black-box property restricts the possible integrations of generated programs to glue-code composition because we only know about the public interfaces of the generated programs. Obviously, this simple glue-code integrative composition prevents the implementation of generators for crosscutting code. Crosscutting code is required to be integrated, or woven, into a program: it crosscuts the implementation of another program. This is in contrast with the fact that many frequently implemented program parts are crosscutting and cannot be implemented as encapsulated programs. In other words, the implementation of crosscutting code generators requires an *invasive integration* of the generated programs instead of simple glue-code integration. The balancing and traversal generators are clear examples of generators of crosscutting code. Although we consider that an integrative composition by glue code is the most simple form of integrative composition, we will always use the term *integrative composition* for an *invasive integrative composition.* Unless explicitly stated otherwise, glue-code integration is not considered when we are dealing with integrative composition.

Structural versus Behavioral Integration

In an integrative composition, the generated programs that are integrated are not independent of each other. On the one hand, there are often required interactions between the integrated programs, while on the other hand, undesired interactions (or interferences) need to be avoided. Once again, this requires that the generators in an integrative composition require solid knowledge about the structure and behavior of generated programs. Hence, integrative composition conflicts once again with the black-box property of generated programs. For example, in the integrative composition of the traversal generator and tree generator, the traversal generator needs to integrate its generated traversal methods in the classes produced by the tree generator. To generate the traversal code, the traversal generator requires knowledge about how to access a child node from within a particular node and which classes implement nodes and which classes implement leaves. Similarly, the balancing generator requires knowledge about the manipulation messages that can be sent to restructure the tree. Last but not least, the methods that are inserted may not override nor overwrite existing methods, as this will cause undesired interactions (interferences).

The previous example is a pure structural integration. The integration of the traversal methods in the tree's classes is a structural modification to those classes. Although it inserts new behavior on the tree, it does not modify existing behavior. In essence the behavior of the tree and the traversal code interacts only through their public interface. Furthermore, the integration only requires other structural knowledge such as the names of the messages and the identification of nodes and leaves. However, in some cases, the generated programs are required to interact by means of more 'overlapping' implementations. For example, in the constraint

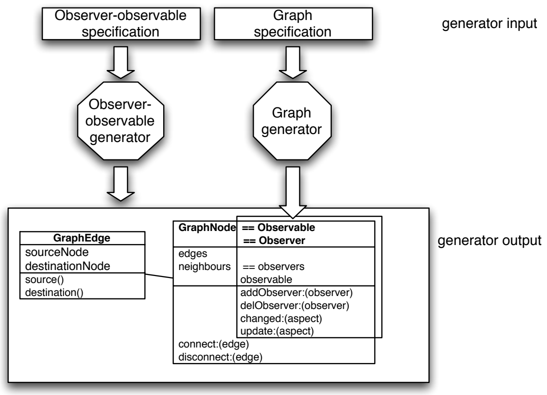


Figure 3.7: Integrative composition of the observer-observable generator and the graph generator.

network generator, we require an integration of the generated programs of the graph generator and the observer-observable generator. The generated classes that represent nodes contain references to their neighbors. The generated framework classes of the observer-observable generator that represent the 'observables' contain references to the observers. In the constraint network, the observers of a node are its neighbors. This integrative composition requires an interaction between the generated programs of the graph generator and the observer-observable generator. Such an interaction can, for example, be achieved by integrating the generated programs such that the list of references to the neighbors is the same list as the list of observers. This can be done by choosing an overlapping implementation for the instance variable that contains the neighbors and the instance variable that contains the observers. This means that the instance variables of both generated programs need to be integrated into one single variable that is used by both.

Figure 3.7 illustrates the integrative composition of the observer-observable generator and the graph generator. Besides a structural integration, this example requires a behavioral integration. The merging of the instance variables gives rise to a behavioral interaction between the generated programs. This interaction implements a behavioral coupling between the two generated programs, i.e. the sharing of the list of observers/neighbors. Of course, this integration is more complex as it requires that both generated programs treat this variable consistently.

3.4 Functional Generator Composition

Now that we have identified two different kinds of composition of program generators, we investigate the appropriateness of functional composition as a technique to implement both kinds of compositions. We will first describe how translation composition can be straightforwardly implemented by functional composition. Afterwards, we explain how integrative composition is problematic if it is implemented as simple functional composition.

We investigate functional composition of generators because every program generator can be represented as a function and a program as its data. In other words, generators are functions over programs. This is because a generator accepts an input program (the input specification) and returns another program (the generated program). The normal way to compose functions is through functional composition. This involves that (part of) the output of one generator is the input of another generator. Although such functional composition is adequate if a *translation* of the generated program is required, it is not appropriate if the generated programs need to be *integrated*.

3.4.1 Translation Composition

Functional composition is particularly adequate for *translation composition*. In its most simple form, functional composition occurs by chaining functions together, i.e. apply a function on the entire result of another function. In practice it will also often occur that we apply a function on a (well identified) part of the result of another function. Evidently, the second function in a composition must understand the result of the first function. Because generators are functions on programs, the second generator must thus understand the generated program of the first generator. As we have described in the previous section, in a translation composition this means that the output language of the first generator is the input language of the second generator.

In many programming languages, this correctness of functional composition can be partially checked by a type checker if a function declares its input and output types. Likewise, we can verify the correctness of a translation composition if a generator declares its input and output languages. The following set of functions[1] shows how the translation composition of figure 3.5 can be expressed using functions and functional composition. The `spec-generator` is declared as a function that accepts a program in the `Grammar` language and outputs two programs, one in the `EBNF` language and one in the `TreeSpec` language. Likewise, the `parser-generator` is a function that converts a program in the `EBNF` language to a `Smalltalk` program. These functions can be composed with the `tree-generator` function that converts a program from `TreeSpec`

[1] We use Haskell [Has] syntax to denote the functions.

to `Smalltalk`. The result of the composition is the `parserAndParsetree-generator` function that produces a combination of two `Smalltalk` programs that implement the functionality described in a `Grammar` program.

```
spec-generator :: Grammar -> (EBNF,TreeSpec)
parser-generator :: EBNF -> Smalltalk
tree-generator :: TreeSpec -> Smalltalk

parserAndParsetree-generator :: Grammar -> (Smalltalk,Smalltalk)
parserAndParsetree-generator(grammar) =
          let (parserspec,treespec) = spec-generator(grammar)
          in (parser-generator(parserspec),tree-generator(treespec))
```

3.4.2 Integrative Composition

As we have described, an integration of generated programs by means of glue-code poses no particular problems. It does not require us to break the encapsulation of the modules in generated programs but does limit the kinds of integrations that can be accomplished. We will first describe how integrative composition by means of glue code is possible using only functional composition. Afterwards, we describe the problems involved with functional composition to *achieve* an invasive integrative composition.

Glue-code Integrative Composition

Building a generator as a composition of other generators that results in integrative composition is particularly easy if the composition of the generated programs merely requires some glue code. Glue code composes separate programs but only uses the public interface of the generated programs and cannot directly change or adapt the internal implementation. The integrative composition thus boils down to invoking the individual generators separately and generate some extra glue code that glues the generated program together. The resulting generator actually produces an integration of the individual generated programs, but because of the encapsulation of the individual programs, no special issues are raised in their integration. Moreover, no knowledge is required about the internal implementation details and thus the black-box property of each generator and generated program is conserved.

Consider, for example, the use of a datacontainer generator and a user-interface generator in the development of a chat application. The datacontainer generator produces a class that implements the datacontainer in accordance with the set of required features. The datacontainer is used to keep a list of open network connections. The user-interface generator produces code that builds a user-interface for the chat application, based on the programmer's descriptions. In the chat application, the generated datacontainer implementation and the generated user-interface implementation only interact through their public interfaces. An example of such an interaction can be found in the method that is executed when the user presses the disconnect button of the chat user-interface. This method will close all network connections by iterating over the datacontainer.

Invasive Integrative composition

Although functional composition of generators is adequate for translation composition and glue-code integration, it is inadequate for *invasive* integrative composition. Of course, we could use functional composition of generators to obtain that one generator *modifies* the program produced by another generator. However, this would mean that a generator always has to accept another generated program and be able to integrate its generated program with it. As we have described before, this requires breaking the encapsulation of generated programs and means that the generator should have detailed knowledge about the generated program it receives as input such as how and where to modify it. As such, during the development of a generator, we need to consider the integration of its generated program in any possible generated program. This introduces a tight coupling between the generators. Changes made to one generator might thus require changes to other generators or otherwise the integration of the generated programs might be broken.

We illustrate this again with a number of functions that represent generators. Consider the following functions that represent the tree, traversal and balancing generators. The treegenerator accepts a `TreeSpec` program and produces a `TreeProgram`. This `TreeProgram` code thus needs to be accepted by both the traversal and balancing generators. The traversal generator also accepts a `TraversalSpec` and produces a `TraversableTreeProgram`, while the balancing generator accepts a `BalancingSpec` program and produces a `BalancedTreeProgram`.

```
treegenerator :: TreeSpec -> TreeProgram
traversalgenerator :: TreeProgram -> TraversalSpec -> TraversableTreeProgram
balancinggenerator :: TreeProgram -> BalancingSpec -> BalancedTreeProgram
```

Using the functions above, we can use standard functional composition to create a traversable tree generator and a balanced tree generator. However, to create a generator that produces a balanced and traversable tree, we need to change either the traversal or balancing generator's signature. Otherwise, the balanced traversable tree generator cannot be built using functional composition of the tree, balancing and traversal generators. For example, consider that we change the balancing generator's function to:

```
balancinggenerator::TraversableTreeProgram->BalancingSpec->BalancedTraversableTreeProgram
```

This required change in signature is not an artificial trick, it reflects an inherent problem in the composition. In essence the first signature of the balancing generator specified that the generator is able to integrate its balancing code correctly into a normal tree, while the second signature specifies that it can correctly integrate the balancing code into a traversable tree. This distinction is important because the balancing generator expects a particular tree implementation to integrate its generated balancing code with. The balancing generator thus has some assumptions (or requirements) about the tree implementation structure to allow a correct integration of its balancing code in the tree implementation. In this example, the possible interferences between the balancing and traversal code are only syntactic (or structural) and easy to prevent. More complex, behavioral interactions and interferences often occur in other integrations. An example of such a behavioral interaction is found in the constraint network generator, as described before. Furthermore, it becomes even more obvious because the observer-observable generator needs to be able to integrate with multiple kinds of programs, and not only with a generated graph implementation. This requires many different possible integrations and possible interferences that must be avoided. Consequently, we must implement a different generator for each possible generated program that can be integrated with our generated program. In the example of the tree generator, a different balancing generator should be written if the balancing code needs to integrate with a traversable tree or a simple tree. This problem is of course applicable to all generators that produce programs that need to integrate with a tree, leading to a combinatorial explosion of different generators that each generate the same or similar programs but integrate it in different generated programs.

Mind that in the context of program generation, `TraversableTreeProgram` cannot simply be defined as a subtype of `TreeProgram`. Subtyping assumes that a `TreeProgram` value (program) is always substitutable by a `TraversableTreeProgram` value (program). This is not the case for generators as they make invasive changes to a program, which require profound knowledge about the entire program in order not to break existing functionality. A generator that assumes that it modifies a `TreeProgram` might thus break the extra traversal functionality defined in a `TraversableTreeProgram`.

In essence, the resulting generators would need to be tightly coupled. This tight coupling will result in a broken modularity when the generators are reused in different composition because their internal implementation will need adaptation to be composable. Therefore, instead of using functional composition of generators, integrative composition requires a different composition technique. The composition must separate the integration from the generation, thereby reducing the coupling between generators and thus rendering them more reusable. On top of this, the separation of the generation and integration forces the generated programs to be more generic (with respect to integrations with other programs) and thus more reusable. Although it is impossible to ensure that any generator can integratively compose with any other generator, the decoupling of generation and integration allows to consider the composition with new generators without necessarily needing to change existing generators.

3.4.3 Analogy with Monad Composition

Because a program generator is a compiler of a domain-specific language, building modular program generators resembles building modular language implementations. Separating the implementation of an interpreter or compiler in functions for each individual language feature has been the focus of research for a number of years in the functional programming community [Ste94]. To accomplish modularity of these functions, monads and monad transformers are used. It is not our intention to explain the particularities of monadic programming here, nor will we use it to actually implement and compose generators. We are describing the composition of monads here to draw the attention to the problems associated with functional composition of monads and the similarity to composition of program generators. For more elaborated information on monadic programming, the interested reader is referred to [Wad90, New].

A monad is a way to structure computations in terms of values and sequences of computations using those values. Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required [New]. This is remarkably similar to building modular program generators, i.e.: the building blocks are program generators, which can themselves be compositions of other program generators. Since monads control how computations (functions) are combined, monads must control the integrative composition of program generators. For that purpose, monads are associated with a 'special' combination function that is responsible for chaining the computations together. In the context of program generators, this combination function implements the integrative composition. As a result, the integrative composition should no longer be implemented in the program generator, but in the combination function. This is exactly what we want.

Consider the following type definitions that respectively represent the traversal and balancing generators:

```
TraversalM p :: TraversalSpec -> (TraversalProgram , p)
BalancingM p :: BalancingSpec -> (BalancingProgram , p)
```

Without delving into monad particularities, the type `TraversalM` is a monad that represents program generators that accept a `TraversalSpec` specification and produce `TraversalProgram` code over some program `p`. This means that the traversal code is integrated in the program `p`. The combination function associated with this monad is able to combine computations that return values of type `(TraversalProgram , p)`. Strictly speaking, the combination function 'threads' the traversal code through all computations (generators). In essence this means that the combination functions handles the integrative composition of the traversalcode on the program. We can now define a generator for traversable trees by defining a function that is built using a tree generator and the traversal generator:

```
generator :: TreeSpec->TraversalM Tree = TreeSpec->TraversalSpec->(TraversalProgram,TreeProgram)
```

The `generator` function above defines a tree generator that accepts a `TreeSpec` and produces a `TreeProgram` while also performing some other computation that produces the traversal code and integrates it into the tree program. The complete signature of this generator thus accepts a `TreeSpec`, a `TraversalSpec` and produces a traversable tree, denoted by `TraversalProgram , TreeProgram`.

We have also defined a `BalancingM` monad in the same way as the `TraversalM` monad. We could think of using both these monads to define a generator function that produces traversable, balanced trees. Such a generator function requires a signature like:

```
generator::TreeSpec->BalancingSpec->TraversalSpec->(TraversalProgram,BalancingProgram,TreeProgram)
```

However, the functional composition of the monads can only lead to:

```
BalancingM(TraversalM p) = BalancingSpec->(BalancingProgram,(TraversalSpec->TraversalProgram,p))
TraversalM(BalancingM p) = TraversalSpec->(TraversalProgram,(BalancingSpec->BalancingProgram,p))
```

Which is not what we want. The type signatures of the composed monads do not correspond with the type required for the traversable, balanced tree generator. Consequently, building a traversable balanced tree generator as a functional composition of the traversal and balancing monads is impossible. In general, the combination of monads is problematic and impossible in the general case [Ste94]. The conclusion is that composition of monads does not work because the composition of monads does not allow a sufficient ‘intertwining’ of the functions.

As a solution to the monad composition problem, monad transformers were proposed [LHJ95]. Composition of monads is done by a monad transformer that accepts a monad and produces a new monad. This means that our previous `TraversalM` and `BalancingM` monads should now be written as monad transformers. The idea is that each monad transformer modifies a monad by inserting its own functionality. In the context of program generators, this means that each program generator actually transforms another generator. The result of this transformation is a generator that combines the generation functionalities and, as such, produces an integration of the corresponding generated programs. We thus represent the traversal and balancing generators as the following monads transformers:

```
TraversalMT m p = TraversalSpec -> m (TraversalProgram , p)
BalancingMT m p = BalancingSpec -> m (BalancingProgram , p)
```

The functional composition of these monad transformers obtains the generator that we desire:

```
generator :: TreeSpec->BalancingMT(TraversalMT(Id)) Tree
      = TreeSpec->BalancingSpec->TraversalSpec->(BalancingProgram,TraversalProgram,TreeProgram)
```

This analogy shows that program generators cannot be simply represented as functions and composed using simple functional composition to obtain a composition of generated programs. Instead, a program generator must *transform* another generator, to obtain a composition of the generators' functionalities. This boils down to a higher-order functional composition of generators. Furthermore, each program generator must provide specific locations in the generated program that can be transformed by another generator without breaking the original functionality of the generator. In monad transformers, this is accomplished by lifting [LHJ95]. Lifting can be seen as a generalization of object-oriented inheritance and overriding [Pre97].

Although the general mechanism of monad transformers provides support for integrative composition of generators, the challenge remains in identifying the possible locations in a program that permit integration and prevent the breaking of existing functionalities in each 'transformation'. In essence, monadic programming could be used to implement the proposed approach to integrative composition, explained in the following chapter, but the identification of the possible interactions remains to be researched. Furthermore, interactions between monads always need to be implemented by lifting the monad operators and the order of composition of monads is important. In this dissertation, we chose not to use monadic programming and developed an integrative composition technique in which the composition order is not important and where integration (and interactions) can be specified in more appropriate terms than lifting.

3.5 Integrative Composition Technique

We have now explained how simple functional composition of program generators is inadequate to accomplish (invasive) integrative composition. The analogy with monad transformers has illustrated that we need to build generators such that one generator modifies another generator. An integrative composition is thus a transformation of one generator by another generator. However, a generator may not modify another generator in undisciplined ways because it might break the functionality of the generator and consequently, also the generated program. In this section, we describe the issues that must be covered and controlled in an integrative composition.

3.5.1 Integrative Composition Interface

To perform an integrative composition, we require knowledge about parts of the generated programs that need to be integrated. In the development of a generator, the developer thus has to foresee where and how other generators might want to integrate their generated programs. Therefore, a generator has to provide an *integrative composition interface* that provides controlled access to the implementation of the generated program such that another generator can specify *where* it can integrate.

In the example generators that we introduced, the tree-traversal generator needs to introduce traversal code into a tree implementation that was generated by a tree generator. The traversal generator produces the required traversal methods and the tree generator produces the classes that implement nodes and leaves. The integration of the traversal methods in the tree classes requires knowledge about what classes implement nodes and what classes implement leaves. The tree generator thus has to open up the black-box property and even the encapsulation of the generated program. Therefore, the generator provides this information about its generated program through an integrative composition interface. Furthermore, the traversal generator needs to know which messages can be sent in each node to access the children of that node. Last but not least, the integration itself requires that we can insert the traversal methods in the generated classes of the tree generator.

Evidently, this controlled breaking of the black-box property and the encapsulation of the generated program needs to allow us to specify an integrative composition of generators in terms of their generated programs. This is described in the following section. Furthermore, modifications to a generated program for the purpose of integrative composition, may not introduce undesired interferences, which is described in a subsequent section.

3.5.2 Integrative Composition Specification

Integrative composition should not be hard-wired in the implementation of a program generator. Instead, an integrative composition of program generators must allow that the *composer* specifies how the corresponding generated programs should integrate. This flexibility is important with respect to reusability of the individual generators in different integrative compositions. It requires a means to specify an integration of the generated programs and it requires that a generator provides sufficient information about the generated program, as described in the previous section.

Of course, an integrative composition specification must remain sufficiently abstract with respect to the generated programs such that the integration specification should not be changed if the input specification of the generator is changed. This is an obvious but important requirement because the generated program of a generator is often different for different input specifications. If an integrative composition were tied to concrete elements in the generated program, then the integrative composition could require changes if the input specification is changed. This is particularly inconvenient if a generator is built as an integrative composition of multiple generators. It would mean that this composed generator would need to change its composition of generators for different input specifications. The integrative composition should thus not be specified in terms of concrete elements of the generated programs. This means that the information provided by a generator about its generated program is not tied to a specific instance of a generated program, but must be applicable to any generated program of the generator. For example, the integration specification of the traversal and tree generators should not be expressed in terms of the concrete names of the traversal methods and tree classes. Instead, the generators need to identify the traversal methods and the node and leaf classes by an abstract term that can be referred to in the integration specification. In this way, the names of the traversal methods as well as the node and leaf classes can change without breaking the integrative composition.

Furthermore, the separate program generators must produce programs that can be integrated to obtain the resulting program. If the balancing generator assumes that the tree will be integrated with is a binary tree, then it is impossible to make the balancing program work on an n-ary tree, without changing the generator. As such, there exist particular assumptions between the different generators in the composition about what they generate. It is clear that we cannot compose anything with anything. Moreover, the more possible functionalities generated by a generator, the more its generated program can be tailored to integrate correctly with other generated programs. For example, if the traversal generator expects that it can generate a program that traverses a tree by calling a method named `getChild()`, then this already compromises its composability.

3.5.3 Composition Conflicts

Because integrative composition breaks the encapsulation of the generated programs, it is likely that certain integrations cause a broken functionality. Therefore, it is desirable to have an automatic conflict detection mechanism for the integrative composition of generated programs. While it is easy to detect and prevent low-level conflicts such as (undesired) method overriding and inadvertent name captures, it is a lot more difficult to detect any higher-level (semantic) conflicts between the generated programs. For example, a particular integration of generated programs causes that both the programs now have to deal with the same datastructure. One generated program might expect a set datastructure, while the other might expect an ordered list. Sets and ordered lists are semantically different datastructures because the set does not store duplicate values, while the ordered list does. If we do not detect such a conflict, caused by their integrative composition, then the individual programs as well as their integration will most likely not work correctly.

3.5.4 Conflict Resolution

Once conflicts are detected, they should be resolved. A developer who specifies the integrative composition has very little knowledge about the implementation of the generated program and no knowledge at all about the implementation of its generator. However, resolution of conflicts does require that the generated program is changed in such a way that the conflict is resolved. This requires appropriate knowledge about the generated program or generator. Because it is undesirable to manually change a generated program, this would mean that we need to change the generated program by changing how it is generated. This can be done by changing the input specification of the generator, or by changing the generator itself. Evidently, changing the implementation of the generator to make the integrative composition work is also undesirable. Furthermore, changing the input specification is not always an option and might not even change the generated program such that the composition conflict is resolved. This implies that conflicts are preferably resolved automatically. This is again particularly easy to do in the case of low-level conflicts, where a simple renaming is often sufficient. However, solving higher-level conflicts is not so easy. In our earlier example, this would mean that the generator needs to adapt its generated program to work with either a set or an ordered list datastructure.

It is desirable that as many conflicts as possible can be resolved automatically. However, it is obvious that the resolution of these conflicts will need to be anticipated when the generator itself is developed. In other words, it is the developer of the generators who knows the internal implementation details of the generated program and the generator. Consequently, it is the developer of the generator who has to specify the resolution of possible composition conflicts. The more potential conflicts a generator anticipates too, the better its composability with other generators.

3.6 Modularization and Composition in Existing Technologies

Some existing generative programming techniques and technologies address the modularization of program generators and the necessary integrative composition. While some generative programming techniques permit us to modularize the implementation of a program generator, these modules are hardly reusable in the implementation of other generators. The modularization techniques are mostly oriented towards structuring the implementation of the program generator and not towards reuse in other generators. In essence, the modules are strongly coupled. This leads to an improved extensibility and maintainability of the program generator, but does not promote reusability of the individual modules. Some generative techniques also support conflict detection if modules are incorrectly composed. However, even if these conflicts can be resolved, this resolution still requires an invasive modification of the generators or the modules.

3.6.1 Ad-hoc and Metaprogramming Generators

Ad-hoc and metaprogramming generators are built using any (traditional) programming technology. The implementation of the generator can thus rely on the existing modularization techniques available in the programming language to modularize its implementation. However, the intention of an ad-hoc generator is not the integration with other generators, but simply to generate its required program. It will thus often be the case that the modularization exists but that there is a strong coupling between the individual modules. Moreover, as any ad-hoc generator can be implemented in a different language and in a different way, the integrative composition can only occur manually. In essence, ad-hoc generators produce their own program and are not designed, nor implemented, with an integrative composition in mind. By definition, they are black-box generators and completely prevent any kind of integrative composition as discussed above.

Metaprogramming or API-based generators are a little different because they operate on the same program representation. This would make it a lot easier to perform integrative composition as their implementation already constructs a program using the same representation and techniques. In essence, the metaprogramming library could protect the generated program from inadvertent breaking of existing functionality. However, this still leaves resolution of composition conflicts up to the developer. It would be possible if the generator can respond to such a conflict by producing alternative code that does not interfere. Mind that this would require a kind of transaction mechanism that checks if the total set of changes made by the generator does not interfere and subsequently applies the changes or 'rolls back' and tries an alternative. Furthermore, some interferences could require that the already existing code is changed, and thus require a roll-back over multiple generators.

Recent macro systems, such as the Scheme (R5RS) macros are powerful transformation systems embedded in a general-purpose language. They support hygienic macro expansion and consequently prevent possible name conflicts automatically. However, more complex interactions and interferences between the generated code of separate macro's are not handled by the system and require a manual implementation in each 'composable' macro.

We can conclude that the mere technique of metaprogramming or API-based generators does not oppose integrative composition. However, an integrative composition does require that the generator is designed and implemented to support the requirements as discussed in the previous section. Therefore, the metaprogramming technique should support this with appropriate features and abstractions. To the best of our knowledge, this is not the case in any of the technologies that can be classified as metaprogramming generators. In contrast, transformation technologies often are equiped with advanced scheduling systems. These transformation techniques are discussed in the following subsection.

3.6.2 Transformational Generators

The basic modularization mechanism in a transformational generator is a transformation rule. The idea is that each transformation rule focuses on a particular part of the transformation process. In theory, it would be possible to accomplish integrative composition by merging the transformation rules of the program generators. However, this approach is especially prone to errors as it accomplishes an implicit integration of the generated programs. In other words, all necessary transformation rules are executed such that both programs are generated but there is no explicit control mechanism that determines how the generated programs are actually integrated. More advanced transformation systems contain a scheduling facility to organize the application order of the transformation rules. This is often required because an incorrect ordering might result in an incorrect generated program. Scheduling of transformations can help us in preventing and resolving errors due to an incorrect application order of transformations but cannot resolve errors caused by the nature of the actual generated code. In essence, because we merge the sets of transformations, we need to find an application order that produces a correct integration of the generated programs. This requires careful analysis of the transformations to come up with a schedule. Transformation systems such as Intentional Programming [ADK^{+}98] provide such a scheduling mechanism. However, the interferences in the integrative composition we discussed above are not caused by an incorrect application order of transformations but are caused by the generated code of individual transformations. Consequently, these interferences cannot be solved by changing the application order of the transformations but should be solved by selecting alternative transformations. In some transformation systems (e.g. Draco [Nei89]), multiple (alternative) transformations can be written that transform the same input element. However, the choice for alternatives is primarily an issue of optimization and driven by input information of the transformation instead of by possible interferences in its output.

3.6.3 Compositional Generators

Compositional generators produce a program by assembling generic modules. This trivially means that compositional generators have a modularization mechanism and can encapsulate the generation of separate concerns in separate generic modules. An integrative composition of compositional generators could be accomplished by merging the set of generic modules. The composed generator can then compose the generic modules of the different generators, leading to an integration of the generated programs. However, this often requires revisions to the generic modules and, as a consequence, profound knowledge on the implementation of each generic module. This is often because the generic modules are not even generic enough and require modifications to their source code to accomplish a correct integrative composition of the generators they are implemented for.

GenVoca generators [BST^{+}94b] are a prominent compositional generation technique that produce their programs by assembling object-oriented layers. Each layer implements a separate feature of the program and can be easily left out of the generated program if it is not required by the GenVoca composition specification. As we have described in the previous chapter, these layers can be implemented with Java mixin-layers [SB98] and C++ templates [Cza98]. In essence the C++ templates allow us to construct mixins in C++. Both implementations are thus more or less the same (at least for the purpose of this discussion). Implementing a GenVoca layer as a mixin-layer complicates the reuse of these layers in totally different program generators. A layer that is implemented as a mixin-layer is designed and implemented for use in a particular program generator. Mixin-layers are not sufficiently generic for integrative composition because they make particular structural assumptions about the other layers in a composition. In essence, a mixin layer is only parameterized with its superclass and possibly also with some constants. As a result, all other implementation details (method calls, variables, method overriding,...) inside the mixin-layer assume that mixin-layer can only be composed with a fixed set of other mixin-layers. The required interactions between the mixin-layers are hard-coded in each mixin-layer (such as method-calls or method overriding for example). It also means that the functionality of that mixin-layer can be easily broken because of changes in the composition or the internal implementation of other mixin layers. Consequently, the implementation of the mixin-layers assumes some *implicit* knowledge about the other mixin-layers in the implementation of the program generator. The composition of the mixin-layers is checked by the GenVoca type checker and more complex dependencies can be made explicit by means of pre- and postconditions on the mixin-layers. These conditions are checked during composition by a design-rule checker [BG97]. Although the mechanism of pre- and post-conditions could be used to detect interferences in an integrative composition, an automatic resolution can only be offered when the conflict occurs because a mixin-layer requires another mixin-layer that was omitted or when the order of the composition is incorrect. All other composition conflicts cannot be automatically resolved because they require a different implementation in the mixin-layers. This is also mentioned and recognized in [BG97]: ‘design-rule checking deals with the testing and assignment of static properties of system designs; it assumes that all transformations are semantically correct’. In essence, an integrative composition technique that merges the set of layers of different generators, violates this assumption. This is because it cannot be guaranteed that the composition of a layer from one generator with a layer from another generator is semantically correct (i.e. a layer transforms the other layer).

It is also interesting to note that mixin-layer composition is remarkably similar to the technique of monad transformers. A composition of mixin-layers results in a complete set of classes. Each mixin-layer can be composed with a set of existing classes and specializes them. In other words, a mixin-layer transforms an existing set of classes and delivers a new set of classes that combines the original classes with the functionality defined in the mixin-layer. This is exactly what monad transformers do: a monad transformer transforms a monad and delivers a new monad that incorporates more functionality. Consequently, the mechanism of compositional generators for integrative composition is appropriate, but the chosen implementation techniques do not offer the right flexibility to perform integrative composition of independently developed modules.

Other compositional techniques such as invasive software composition [Ass04] and aspect-oriented programming [KLM$^+$97] restrict the possible integrative compositions and consequently limit the possible interactions between the generated programs. For example, in aspect-oriented programming languages, the integration of the aspects in a program can only occur through advices and method introductions. Furthermore, if a composition conflict is detected, the developer is required to manually adapt the aspects (e.g. order of advices and method names of introduced methods). In subject-oriented programming [HO93, OKK$^+$96] or multidimensional separation of concerns [OT99], the different composition rules allow to integrate programs in many different ways. Nevertheless, in the case of a composition conflict, the developer is required to adapt the program himself. Although this is done automatically for the simple syntactic naming conflicts, other conflicts (such as typing conflicts) require a manual adaptation of the separate programs.

3.7 Conclusion

In this chapter we provided a characterization of program generator composition with the intention of building modular program generators. We have explained how integrative composition of generators is required such that the generated programs are integrated. Moreover, we have shown that this involves that generators need to be composed by transforming the other generator. Achieving this in a flexible way involves some issues such as integration specifications, automatic conflict detection and resolution. We conclude that no satisfactory means exists to express integrative composition of program generators. In the following chapters, we introduce a generative technique to implement and execute integrative composable generators. An appropriate implementation technique for integrative composable generators is presented as an extension to the logic metaprogramming technique.

Chapter 4

Building Integrative Composable Generators

Without delving into concrete implementation details, this chapter describes the generative programming technique for the implementation and integrative composition of program generators. This generative programming technique addresses the requirements of integrative composition that were identified in the previous chapter. Because this technique could be implemented on top of different kinds of generative programming technologies, we defer a concrete implementation by means of logic metaprogramming to the following chapter.

4.1 Introduction

In the previous chapter, we have described that integrative composition is required to allow the modularization of program generators according to the different concerns that need to be generated. The integrative composable generators, that each generate a separate concern, can then be re-used in the implementation of other generators or as stand-alone generators. The development of such integrative composable generators requires an adequate generative programming technique that supports their implementation and composition in accordance with the requirements identified in the previous chapter. In this chapter, we describe how the generative programming technique that is developed in this dissertation tackles these requirements. In the proposed technique, a program generator essentially consists of multiple generative programs that each produce a well-defined part of the generated program. What these *program parts* are, depends on the output language of the generator and on the implementation decisions taken by the generator developer. Some of these program parts are exposed through an *integrative composition interface* and can be integrated in another generated program. Such an integration is specified using an *integration specification*.

The program generators can automatically adapt their generated programs to achieve an integrative composition specified in an integration specification. Program generators can also adapt their generated programs to avoid detectable composition conflicts. These conflicts are automatically detected by the generative system and force the program generators to resolve the composition conflict.

The overall integrative composition mechanism is language independent and the generative system can be configured easily to allow integrative composition of program generators with various, but identical, output languages. In the case of program generators that produce programs in domain-specific languages, a domain-specific integrative composition can be specified. Such a domain-specific integrative composition is often more appropriate because it is specified in the domain-specific terms. Moreover, a domain-specific integrative composition provides the opportunity to detect and resolve *domain-specific composition conflicts*.

This chapter is organized as follows. In the next section, we describe the overall architecture of the generative technique. We explain the important parts of our technique and how they relate to each other. Section 4.3 describes the actual integrative composition mechanism in detail, followed by section 4.4 that describes how integrative composable generators are implemented. Section 4.5 explains the important elements of the generative system that executes the generators. Section 4.7, describes how we can realize domain-specific integrative compositions and a composition of composed generators. Finally, section 4.8 discusses the development methodology for integrative composable generators using the technique described in this chapter.

4.2 Architecture

The following subsections provide an overview of the *implementation* of integrative composable generators, followed by a description of the overall architecture of the *generative system* that enables the building of such generators.

4.2.1 Implementing Integrative Composable Generators

The generative technique presented in this dissertation contains three important elements: language definitions, generator implementations and integration specifications. Consider the schematic illustration of a modular integrative composition in figure 4.1. The figure shows an integrative composition of generators `X` and `Y`. These generators accept programs $\mathtt{P}_x$ and $\mathtt{P}_y$ in languages $\mathtt{L}_x$ and $\mathtt{L}_y$ respectively and both produce a program in language $\mathtt{L}_a$. They are involved in an integrative composition that establishes the integration of their generated programs `A1` and `A2`. The integrated program

`A1 + A2` is further translated into language $\mathtt{L}_z$ by generator `Z`. Consequently, generators `X` and `Y` are also involved in a translation composition with generator `Z`. In this illustration, we can identify three different conceptual elements in the modular implementation of program generators. These elements are:

- the *internal implementation* of the program generators (`X` and `Y`), built to allow an integrative composition.
- the *integration specification* (`S`) that establishes an integrative composition of the generators `X` and `Y` and, consequently, the integration of the generated programs `A1` and `A2`. The integration specification is specified in terms of the *integrative composition interfaces* of both generators.
- the *languages* ($\mathtt{L}_x$,$\mathtt{L}_y$ and $\mathtt{L}_a$) that are defined by the individual program generators.

We will now describe each of these elements and their relations in more detail.

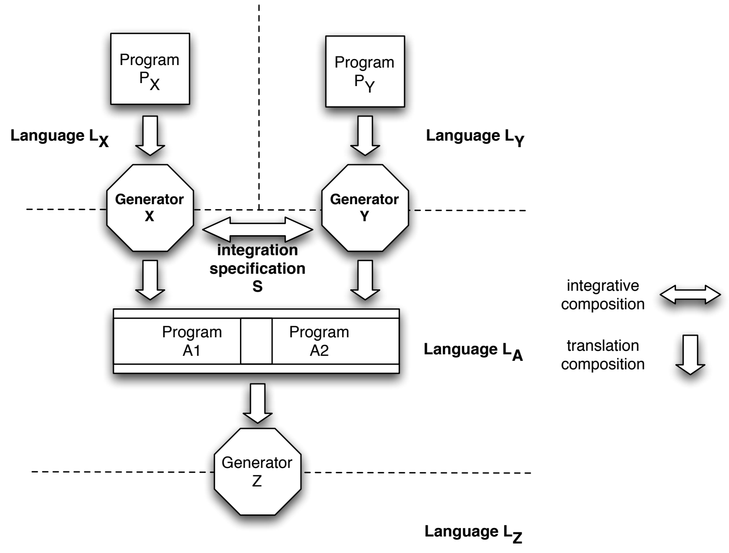


Figure 4.1: Conceptual overview of an integrative composition.

Integrative Composition Interface and Integration Specification All integrative composable generators define an *integrative composition interface.* The integrative composition interface identifies and exposes separate *program parts* of the generated program for use in an *integration specification*. An *integration specification* defines an integrative composition of generators and consists of a set of *integration relations* that each specify an integration of the separate program parts (identified in the integrative composition interfaces). The integrative composition interface is defined by the developer of the generator, while the integration specification is defined by the developer who composes the generators. The definition of an integration specification requires knowledge on the output language of the program generators and requires limited knowledge about the generated programs themselves. This limited knowledge about the generated programs is exposed through the integrative composition interfaces of the program generators.

Generator Internal Implementation The internal implementation of a generator is a set of *generative programs* that each produce a *program part* of the entire generated program. These generative programs implement the generation of the program parts as well as the *resolution of composition conflicts* and the necessary *adaptations* that need to occur to the generated program in an integrative composition. In addition to the generative programs, the internal implementation of a generator also contains the definition of a set of *dependency relations* between the program parts. These dependency relations ensure that the complete program is adapted consistently. As a result, given an integration specification, a generator can adapt its generated program to fit the integration of the generated programs. The adaptation is implemented by the developer of the generator and is automatically enforced by the generative system. A developer is able to implement these adaptations because the possible integrations and composition conflicts are delimited in the definition of the language that is used by the generator as its output language.

Language Definition Each generator 'defines' an input language. For example, in figure 4.1, generator `X` 'defines' language `Lx`. This does not only mean that the generator translates programs in that language. The generator also includes an entire definition of the *syntactic* elements of the language. This definition includes how a program is represented and divided in separate program parts. It also contains an explicit definition of the possible integrations that can be expressed and the possible composition conflicts in that language. Besides the languages defined by each generator, a definition of the most low-level output language is also included in the generative system. Consequently, all languages that can be used to implement generated programs are defined in a language definition. As a result, a developer that implements a generator (with a particular output language) knows about the possible *statically* or *syntactically* detectable conflicts and integrations that might occur in an integrative composition of that generator. In other words, the language definition governs the possible integrative compositions of generators that produce programs in that language. The definition of a language will be treated as a separate identity in the overall architecture of our generative programming technique.

The definition of languages renders the generative technique for integrative composable generators independent of a particular output language of a program generator. The generative system allows to define and use various languages for the implementation of the generated program. It is particularly interesting to define domain-specific languages because they allow to describe domain-specific integrative compositions. *Domain-specific integrative compositions* are often more appropriate then integrative compositions of low-level generated programs. First of all, the integrative composition can be specified in domain-specific terms instead of in low-level implementation terms. Second, a domain-specific integrative composition allows to detect and possibly resolve domain-specific composition conflicts. The detection of such conflicts can prevent semantic interferences between the generated programs, as opposed to the syntactic conflicts that can be detected in the low-level executable language.

4.2.2 The Generative System

The implementation of the generator, the definition of the language and the specification of an integrative composition are handed to the generative programming system that executes the generators and performs the integrative composition as specified in the integration specification. This generative system also consists of three conceptual parts:

Generative Programming Language An important element of any generative system is the language in which a generator is implemented. The generative programming language for integrative composable generators supports the definition of an integrative composition interface and the implementation of the adaptations to achieve a correct integrative composition. In this dissertation, the generative programming language is based on logic metaprogramming and will be introduced in the next chapter.

Conflict Detection Mechanism The generative system checks all generated programs for the occurrence of composition conflicts with other generated programs. These composition conflicts are defined in the language definition of the implementation language of these generated programs. Since the language definition contains only all syntactic elements of a language, the possible composition conflicts are limited to those that can be detected statically. If a composition conflict occurs, the system automatically forces the generators to adapt their generated program to resolve the composition conflict. If no resolution can be offered, the composition fails.

Integration Enforcement The integration specification describes an integrative composition and enforces the integration of the generated programs. On the one hand, the integration specification is used by the generator to produce an appropriate program that can be integrated. On the other hand, the specification is used by the generative system to enforce and verify that the separately generated programs are effectively integrated as it is specified.

Roadmap

In the remainder of this chapter, the individual elements of the overall architecture that we introduced above are explained in detail. In the following section (section 4.3), we describe the integrative composition of program generators. We explain what program parts are, how an integrative composition interface is defined and what an integration specification is. The implementation of integrative composable program generators themselves is discussed in section 4.4, followed by the description of the generative system that executes the implementation and composition of program generators in section 4.5. Because the language definition 'configures' all other parts of the generative technique, it is explained where it is appropriate. Section 4.6 summarizes the individual elements of a language definition. For simplicity, we will illustrate the implementation of integrative composable generators with program generators that produce Smalltalk programs. The use of generators that produce programs in domain-specific (modeling) languages is discussed in section 4.7.

4.3 Integrative Composition

To reconcile the advantages of black-box generated programs with the possibility of integrative composition, each generator defines an integrative composition interface that exposes the generated program as a list of separate program parts. The integrative composition interface thereby breaks the black-box character of the generated program. This interface also allows a controlled access for integration purposes to these separate program parts. We will first describe what these program parts are, followed by what an integrative composition interface is and how it exposes these program parts. Afterwards, we explain how an integration specification is specified.

4.3.1 Program Parts

Program parts are individual pieces of a program, or more specifically, of the generated program's parsetree. The composition interface exposes separate parts of the generated program's parsetree such that an integration can be specified in terms of these parts (which is described later on). The kinds of program parts that can be exposed are different for each possible output language. Therefore, each language definition defines the possible parts of a program's parsetree that can be exposed through an integrative composition interface. For example, a generator that produces object-oriented programs can expose separate methods, variables, classes, statements, etc... as program parts. The actual technical representation of these program parts depends on the implementation technology of the program generators, and is not discussed in this chapter. However, the representation of each program part needs to contain sufficient information to identify the location of the program part in the entire program's parsetree. Furthermore, the assembly of the separate program parts must result in the complete generated program. To achieve this, any given program part needs to contain information such that its parent in the parsetree representation of the program can be uniquely identified. This means, for example, that method or variable definitions need to declare the class in which they are defined. As a consequence, the location of each program part in the entire parsetree can be uniquely identified and the entire generated program can be reconstructed as an assembly of the individual program parts.

Each program part of the generated program is identified by a unique name. Furthermore, a program part also has a type that corresponds to the type of the program part in the parsetree. The possible types are consequently defined for each language in which a program can be generated. A program part is thus a tuple `(name,type,representation,internals)` where

name is a unique name that identifies the part.

type is the kind of the program part.

representation contains a part of the program code included in the program part that is exposed. This must include information to uniquely determine the program part in the entire generated program.

internals Contains the rest of the program code of the program part. It remains a hidden piece of the internal implementation of the generated program.

The difference between the representation and the internals is determined in the definition of the language. It can be used to determine that a particular piece of a program part is always part of the hidden internal implementation of a program. For example, a program part that represent a method can hide the method body in its *internals*.

Smalltalk Program Parts

The example language that we will use throughout this chapter is the Smalltalk object-oriented language. For Smalltalk, we have defined that class definitions, method definitions and variable definitions can be exposed as separate program parts. Consequently, the possible program part types are `class`, `method` or `variable`, respectively. We have also confined ourselves to a simplified representation of a Smalltalk program. We have, for example, confined ourselves to the representation of the instance variables and methods. We have also omitted a number of technical specifications that can be included in a class definition. These restrictions have no impact whatsoever on the generative programming technique and were only made to simplify the given examples. It would, for example, be trivial to extend this representation to include class variables, class methods, shared variables, etc... in the representation. The details of the program parts are:

class A `class` program part contains a class definition. A class definition only contains the class name and the superclass name in its representation. No hidden internals are included in this part.

variable A `variable` program part contains an instance variable definition. The representation contains the variable name and the class in which it is defined. A `variable` program part can optionally describe the type of the variable. The type of a variable is often valuable information in an integration, although it is not required to generate Smalltalk programs since Smalltalk is a dynamically typed language. This program part does not contain hidden internals.

method Each `method` program part defines an entire instance method. The representation consists of the method's name, its signature and the class in which it is defined. The entire method body, containing all statements and local instance variables, remains hidden in the internals of the program part.

A Smalltalk program that is represented as a set of such separate program parts can be entirely reconstructed from the information contained in these program parts. Each program part contains sufficient information to determine it as a unique part of the entire generated program.

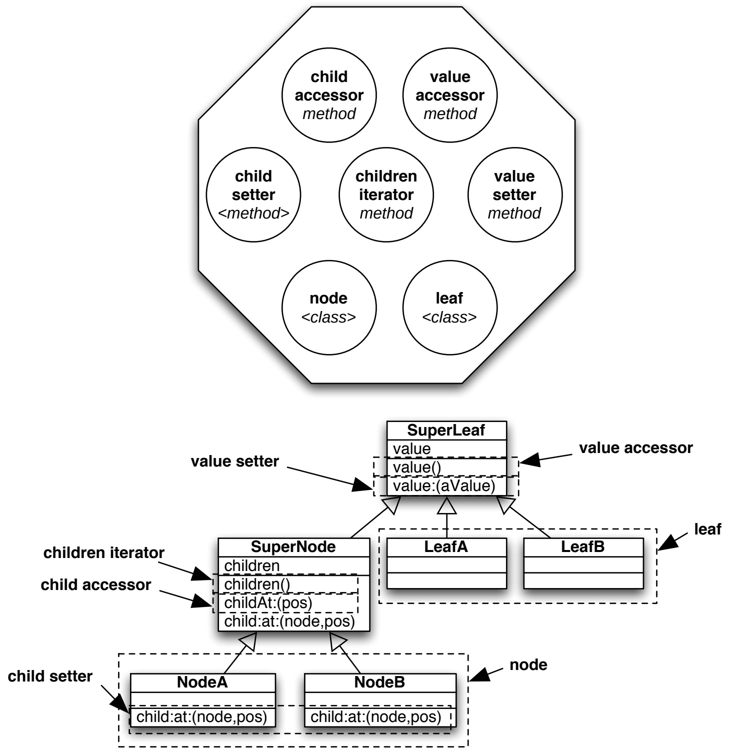


Figure 4.2: Schematic view of the tree generator and its integrative composition interface.

4.3.2 Integrative Composition Interface

An integrative composition interface is a *list of program parts*. We will refer to the program parts exposed in an integrative composition interface as 'public' program parts. A public program part is identified in the integrative composition interface by its name and type. It is also important to note that not all program parts of a generated program are public parts. The developer of a program generator exposes only those parts as public parts that are deemed useful in the context of an integrative composition. Other program parts remain hidden. We will refer to these hidden program parts as 'private' program parts.

Figure 4.2 gives a schematic overview of the tree generator and its integrative composition interface. The generator is displayed in an octagon and the integrative composition interface is shown by means of circles. Each circle represents a public program part and contains the name (in bold) and type (in italic) of that program part. The figure also contains a schematic representation of the generated tree implementation in which we also identified the public program parts. We will use this schematic notation to represent generators and their integrative composition interfaces throughout the remainder of this dissertation.

Figure 4.2 shows that the tree generator exposes the classes that implement the nodes and the leaves of the tree as public program parts. It is desirable to expose these classes as public program parts because an integration of the generated tree implementation with another program will often require the insertion of additional methods and variables in these classes. Therefore, the developer has chosen to expose these program parts in the integrative composition interface. It is also important to distinguish the node classes from the leaf classes. In essence, additional behavior that is integrated in the tree implementation will often distinguish between the nodes and the leaves of the tree and consequently require to integrate different methods in the node and leaf classes. Furthermore, a method that is integrated in a tree's node class might need access to the child nodes and the value contained in the node. Therefore, either we expose the variables that contain the child nodes and the value as public parts or we expose some getter and setter methods for these variables as public parts. In the latter case, the variables remain private program parts. This choice is up to the developer of the tree generator because it depends on how the variables must be used. In this implementation of the tree generator, we remain close to an entire encapsulation of the generated program and do not expose the variables in the composition interface. The result is that in this generator, all program parts available in the public interface of the generated tree implementation itself are also public programs parts in the generator's composition interface. Many internal implementation details about the variables and the method bodies remain hidden. These internal details will be shown later on. In figure 4.2, some types of parts are enclosed between <and >. It means that the part actually represents a list of program parts of that particular type. The representation of such lists of program parts is explained in the following subsection.

We have now described how an integrative composition interface breaks the black-box property of a generated program by exposing parts of the generated program as public program parts. However, the integrative composition interface does not completely break the black-box property of the generated program because it only provides a limited insight (for integration purposes) in the internal implementation of a generated program. The distinction between public and private program parts makes sure that many internal implementation details of the generated program remain hidden. Furthermore, the granularity of the chosen program parts also determines the level of detail that can be exposed via the composition interface. For example, for Smalltalk programs, individual statements of a method body cannot be identified as separate program parts. Last but not least, it is important to note that the encapsulation of the generated program is also broken because the possible program parts are not limited to the encapsulated modules provided by the programming language. In our example in the Smalltalk language, we are able to expose private instance variables and methods as public parts in a composition interface. These program parts are not encapsulated modules in the Smalltalk language but are treated as separate program parts in the generation process.

4.3.3 Special Program Parts

Besides 'normal' program parts, two special kinds of program parts can be included in an integrative composition interface. One special kind of part is a program part that represents a list of program parts. Another kind of program part is a *required* program part.

Program Listparts

Although each generator specifies the separate program parts of any of its possible generated programs, it does not mean that each of those generated programs includes the same number of parts. Depending on the input specification, a program generator may need to produce a different number of program parts. A possible generated program may thus include parts that are not included in another possible generated program of the same generator. For example, in the case of a tree generator, the number of generated classes that represent nodes and leaves clearly depends on the input specification. Therefore, in figure 4.2, the parts that represent the nodes and leaves actually expose a list of classes. A single program part name can thus represent a list of program parts. Such a program part is referred to as a *program listpart*. This makes it possible that a generator produces none, one or more parts that fulfill a particular role, depending on the input specification of the generator. A program listpart is distinguished from normal, 'singleton' parts because it has a different type. The type of a listpart is noted as the type of the program parts it contains, surrounded by brackets (e.g. `<class>`). Omitting this special kind of program part would limit the power of a program generator as all generated parts would be statically fixed in the implementation of a program generator. As a consequence, all generated programs would consist of the same number of parts and can only differ in the internal implementation of these parts. The use of program listparts provide the extra flexibility to produce a variable number of parts.

Required Parts

In some cases, it is useful to specify 'empty' program parts in the composition interface of a program generator. They are 'empty' because the generator does not provide an implementation for them. Instead, these parts represent required parts that must be produced by another generator. The generator that defines such required parts in its integrative composition interface requires that these program parts are 'filled in' using an integrative composition. Without them, the program generator cannot produce a correct program. A required program part can also impose a number of restrictions on the actual part that it can integrate with. This will be explained when we describe the internal implementation of a program generator.

The required parts are often used in the development of crosscutting code generators. Such generators always require an integrative composition with another generator to produce a correct program. The crosscutting code generator will integrate its crosscutting code in the required parts that are identified in its composition interface. We will describe the use of such required parts in the following section where we describe the integrative composition of the tree generator and the traversal code generator. Thoughout the dissertation, required program parts are represented as a circle with a dashed contour line.

4.3.4 Integration Specification

An integration specification is a set of *integration relations* between public program parts, exposed by different program generators. A single integration relation specifies a particular integration of the program parts that it is applied to. The combination of all integration relations determines the actual integration of the entire generated programs. The possible integration relations that can be used to specify an integrative composition are specific for each possible output language of a generator. Therefore, the possible integration relations are defined for each language.

For example, the integration relations available in the Smalltalk language are relations to specify that methods or variables integrate in a class, that classes or variables unite or that methods combine in a particular order. These integration relations are schematically depicted in figure 4.3. In the picture of each integration relation, the program part at the left hand is implemented by a different generator than the program part at the right hand. Each integration relation has a *direction.* Consequently, we can speak of an *origin* and *destination* program part of the relation. Most integration relations are uni-directional but some integration relations are bi-directional. We describe each of the integration relations for the Smalltalk programming language in more detail:

In The `in` integration relation can be established between a `method` or `variable` program part (the origin) and a `class` program part (the destination). It expresses that the corresponding `method` or `variable` definition needs to be implemented *inside* the `class`.

Subclass The `subclass` integration relation is used to express an integration of `class` program parts by subclassing.

Overrides The `overrides` integration relation defines that one `method` part (the origin) overrides another `method` part (the destination). It consequently enforces that the methods share the same signature and are implemented in classes in the same inheritance hierarchy.

Unite The `unite` integration relation can be specified between program parts of the same type. It specifies that these program parts need to be exactly the same and represent only a single program part in the integrated generated program.

IncludeBefore & IncludeAfter The `includeBefore` and `includeAfter` integration relations can be established between methods that share the same signature. They define a method combination where the destination `method` part's body is included before or after the origin `method` part in the integration relation.

More complex integration relations, such as those introduced in subject-oriented programming and multidimensional separation of concerns [OT99, TOHJ99], could also be considered but were not implemented in the context of this dissertation.

For each integration relation specified in an integration specification, the generative system enforces that the generators produce an implementation for the program parts that adheres to that integration relation. The generative system verifies the generated program parts and the composition fails if the program parts do not adhere to the integration relation. The description of how program parts satisfy an integration relation is included in the language definition, together with the definition of the integration relations. Consequently, the generative system only verifies and enforces the integration relations. The actual adaptations of the generated program parts that are necessary to adhere to an integration relation are made by the generator itself. The developer of a generator thus needs to implement the necessary adaptations to the generated program parts so that they can adhere to the possible integration relations. How this is done will be explained when we describe the internal implementation of a program generator. It is possible because all possible integration relations between program parts are defined for each language.

Integration of Listparts

Integration relations between listparts are enforced between the individual parts contained in the listparts. However, we do not enforce that both listparts must contain an equal number of program parts. Therefore, an integration relation on listparts enforces that all program parts that are contained in the originating listpart of the

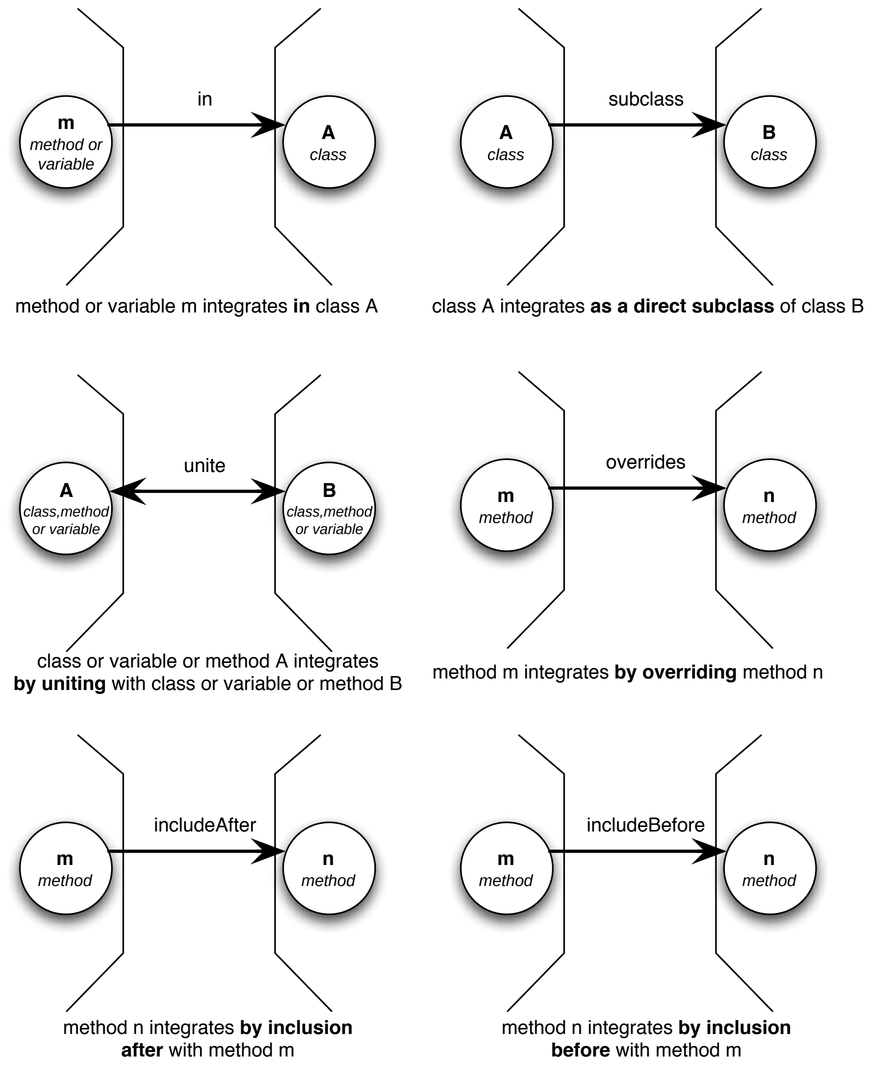

in
m
method or variable
A
class
method or variable m integrates **in** class A
subclass
A
class
B
class
class A integrates **as a direct subclass** of class B
unite
A
class,method or variable
B
class,method or variable
class or variable or method A integrates **by uniting** with class or variable or method B
overrides
m
method
n
method
method m integrates **by overriding** method n
includeAfter
m
method
n
method
method n integrates **by inclusion after** with method m
includeBefore
m
method
n
method
method n integrates **by inclusion before** with method m

Figure 4.3: Possible integration relations between generated parts of Smalltalk programs.

relation must adhere to the integration relation with a program part contained in the destination listpart of the relation.

Integration of Required Parts

In an integration specification, the integration of required program parts is not different from the integration of other program parts. The required part needs to be integrated (most commonly by means of a `unite` integration relation) with a program part produced by another generator, thereby fulfilling the requirement.

Example Integrative Composition

An example of an integrative composition specification is shown in figure 4.4. The traversal generator produces traversal methods for both leaves and nodes. The integrative composition that is shown establishes an integration of the generated tree and traversal programs. More specifically, the traversal methods for nodes are integrated into the classes that implement the nodes. Likewise, the traversal methods for leaves are integrated into the classes that implement the leaves. Since both generators expose these parts and both generators produce Smalltalk code, we can specify an integrative composition. This integrative composition also immediately shows the use of the required program parts and the listparts.

The traversal generator includes a required program part in its integrative composition interface. This required program part describes that the program generator needs the method that allows to access the child nodes from within a node. This required program part must consequently be 'filled in' with the appropriate method program part that implements this access method. Therefore, a `unite` integration relation is drawn between the required part and the method part produced by the tree generator that implements this method. Of course, this requires that the generated traversal implementation is designed to accommodate the specific method signature of this method part. We will delve into these details when we discuss the internal implementation of a program generator. Another interesting aspect of this example integration specification is that the `in` integration relations actually integrate two listparts instead of two simple parts. For example, the node-traversal methods listpart represents a list of traversal methods for nodes. Each of these traversal methods must integrate with its corresponding node class. The `in` integration relation enforces that the generation of the `node traversal` methods listpart produces a listpart that contains `method` program parts that each integrate in a corresponding node class. This is because the generation of the `node traversal` methods takes the presence of an integration relation into account to produce its program part. This is explained when we describe the internal implementation of a program generator

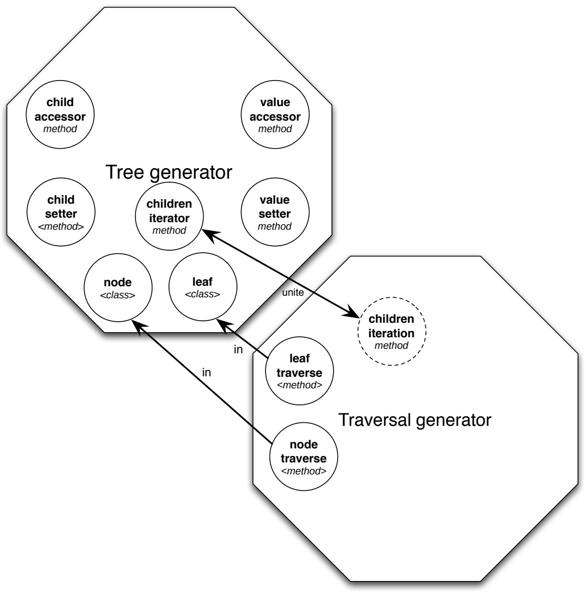


Figure 4.4: Integration relations between the tree and traversal generators.

4.4 Generator Implementation

Until now, we have only described how an integrative composition between generators can be specified. In this section, we focus on the internal implementation of the generators to make integrative composition possible. In particular, we describe how the internal implementation deals with 'integrative variabilities'. In the following subsection, we explain what these integrative variabilities are and subsequently describe the overall implementation technique that facilitates their implementation using *dependency relations* and *alternative generated program parts.*

4.4.1 Integrative Variabilities

Generative programming techniques provide linguistic support for the implementation of a program generator in terms of the commonalities and variabilities of its possible generated programs. In contemporary program generators, these variabilities are determined by the possible input specifications and result in different functionalities implemented by the generated programs. Consequently, contemporary generative techniques provide support for the implementation of such 'functional' variabilities. In addition, a generative technique that supports integrative composition, in accordance with the requirements identified in the previous chapter, has to provide support to address the variabilities that arise from the possible integrative compositions. These 'integrative' variabilities are the adaptations of the generated program to enable its integration with another generated program. This includes the necessary adaptations to correctly integrate the generated programs and to resolve specific composition conflicts. 'Functional' variabilities are very different from 'integrative' variabilities. While the former arise from differences in the input specification, the latter arise from required differences in the output program to establish an integration with another program. 'Integrative' variabilities are not about the implementation of *different* functionalities provided by the generated program. Instead, 'integrative' variabilities are about implementing the *same* functionalities using a different generated implementation.

Examples

In the implementation of a generator that produces a data container implementation, a possible example of a functional variability is about the size of the data container. The generator can, for example, produce a fixed size data container (e.g. an array) or it can produce an implementation of a data container that can grow dynamically in size. Such a functional variability is determined by the input specification of the generator and is very different from the possible integrative variabilities. The name of the implementation class and its method names are example integrative variabilities for a data container implementation in an object-oriented programming language. This is because another class with the same name can be generated by another generator, leading to a name conflict. Furthermore, a particular integrative composition can enforce that the generated implementation class is a subclass of another (generated) class. This integration requires us to deal with integrative variabilities in the class definition and the names of the methods defined in the class. Although these example integrative variabilities are rather simple, it is important to deal with them explicitly as well. For example, a name can be part of the public interface of a generated program, where want a decent name and not an automatically renamed identifier. A more complex integrative variability can be identified in the example of the integration of the generated graph implementation and the observer implementation, introduced in the previous chapter. This integration requires a `unite` integration relation between two `variable` program parts. However, each of these variables can have a different type. To achieve a correct integration, both generators need to adapt their generated programs such that they agree on the variable type, if one is supplied. This is an integrative variability that requires a substantial difference in the implementation of the generated programs, i.e. the use of a different type for the variable and all generated program code that uses it.

Determining the Integrative Variabilities

The integrative variabilities that need to be handled in the implementation of each integrative composable generators are determined by:

Integration Relations As we have explained, an integrative composition is specified by means of integration relations between public parts of different generators. The implementation of the generator needs to deal with the possible integration relations because it needs to be able to generate a correct implementation that adheres to the integration relations.

Composition Conflicts Some integrative compositions result in undesired interferences between the generated programs. In such a case, a composition conflict is detected and the generation of the program actually fails. However, whenever possible, an integrative composable generator should anticipate the possible composition conflicts and try to resolve them.

We already described the possible integration relations in the Smalltalk language. The possible composition conflicts that can be detected in the integration of Smalltalk programs are: duplicate class names, duplicate method names in the same class, duplicate variable names in the same class, accidental overriding of methods and shadowing of variables. Most of these conflicts are actual Smalltalk language conflicts, but the accidental overriding is a specific composition conflicts. We describe later on how these composition conflicts are actually detected. We will now describe how the internal implementation of a program generator can resolve these composition conflicts. In essence, the implementation of integrative variabilities is the most important aspect of the internal implementation of integrative composable program generators. We deal with the integrative variabilities through the definition of dependency relations between the individual program parts of a generated program, through separate generative programs for each program part and through the parameterization of the generative programs with the integration and dependency relations. All these elements are explained next.

4.4.2 Dependency Relations

There are particular dependencies between all parts (public and private) of a generated program that must remain valid in all integrative compositions. To implement and enforce these dependencies, the developer of a generator needs to define *dependency relations* between all program parts. The generative system enforces these dependency relations in all possible generated programs of the generator.

Figure 4.5 shows the private and public program parts in the implementation of the tree generator, as well as all dependency relations imposed between them. The private program parts are illustrated with gray-filled circles, as opposed to the public programs parts that are printed as white circles. The private program parts of the tree generator contain the definition of some common superclasses for all generated leaf and node classes. These common superclasses define the variables that are used to contain the value and child nodes in a particular node or leaf. Figure 4.5 also shows this implementation structure of the generated tree implementation.

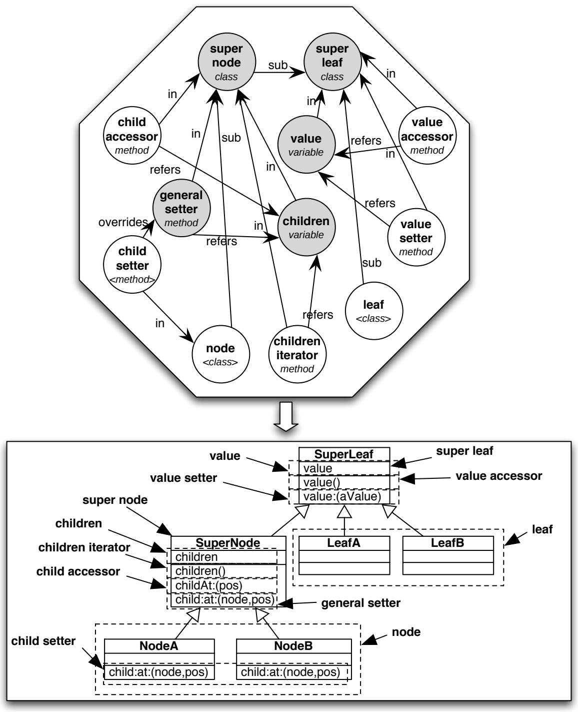


Figure 4.5: Relations between the public and private parts in the tree generator.

One may notice that many dependency relations are in fact integration relations. This is because dependency relations are technically identical to integration relations. Dependency relations enforce a dependency (which can be an integration) between the program parts that they are imposed on. The possible kinds of dependency relations that can be used also depends on the output language of the program generator. Therefore, the possible set of dependency relations is specified for each possible language and also frequently overlaps with the possible set of integration relations. However, we will always refer to *dependency relations* to denote the relations used inside a program generator.

Smalltalk Dependency Relations

For the generation of Smalltalk programs, we have defined that all integration relations, except for the `unite` integration relation can be used as a dependency relation. The additional dependency relations in the Smalltalk language are shown in figure 4.6. We can express `refers`, `contains`, `self-calls` and `calls` dependencies between generated parts:

Refers & calls The `refers` and `calls` dependency relations are similar. The `refers` relation states that a particular method refers to a particular variable by its name. The `calls` relation states that a particular method calls a particular other method. These dependency relations encode the dependencies that the method bodies need to refer to the correct variable or method name. The dependency relation can be enforced because a method program part explicitly declares which variable or method names it uses.

Contains The `contains` dependency relation defines that a particular variable contains a value of a particular class. In other words, the variable is typed to contain an instance of that class.

Self-calls The `self-calls` dependency relation is an extension of the `calls` relation. It does not only enforce that the first method calls the second method in the relation, it also enforces that the methods be defined in the same inheritance hierarchy.

The explicit definition of these dependency relations in the implementation of a program generator prevents the breaking of these dependencies in particular integrative compositions. Furthermore, these dependencies can also automatically trigger the definition of additional integration relations in an integration specification. This is discussed later on in section 4.5.1. It is also important to note that particular other dependencies between generated program parts can be expressed using *generator-specific* dependency relations. The dependency relations that can be used inside a generator are thus not limited to the general language-specific dependency relations. While the language-specific integration and dependency relations are defined in the language definition, the generator-specific dependency relations are defined inside a generator.

4.4.3 Generative Programs

Each separate program part is generated by a separate generative program. The internal implementation of a program generator thus contains a generative program for each private and public program part. This is depicted in figure 4.7. Each of

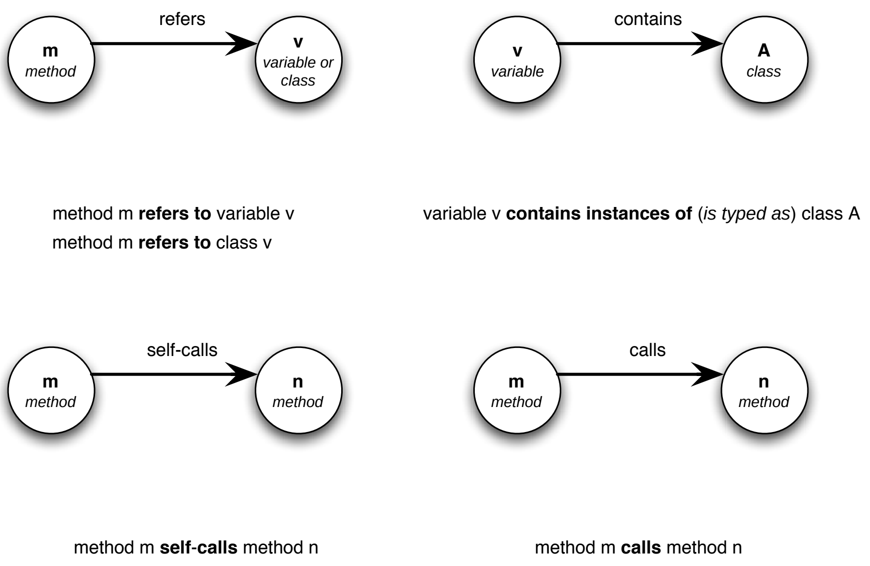

refers
m
method
v
variable or class
contains
v
variable
A
class
method m **refers to** variable v
method m **refers to** class v
variable v **contains instances of** (*is typed as*) class A
self-calls
m
method
n
method
calls
m
method
n
method
method m **self-calls** method n
method m **calls** method n

Figure 4.6: Dependency Relations in Smalltalk.

these generative programs focuses only on the generation of their particular program part. This forces the generator developer to explicitly parameterize the generation of that part with all information that is required from the generation of other parts to produce its own program part. These other program parts are exactly those program parts in the destination of an integration or dependency relation that originates from its own part. In other words, a generative program for a program part is *parameterized* by the generated program parts it needs or depends on. Each generative program can retrieve the necessary information from the program parts it depends on because the generative system provides access to all relations and their destination program parts.

For example, as shown in figure 4.7, the generative program of a method that `refers` to a particular instance variable of the class it is located `in`, requires information from the generation of that variable definition and the generation of the class definition. This information will certainly include the name of that variable and class. This is because the method can only be correctly generated if it uses the correct name for the variable. Consequently, it needs this information from the variable program part. Similarly, the generation of the method requires the name of the class. While this latter parameterization might seem strange, we do require this because we specifically enforced that a method definition program part must declare all information needed to locate it in the entire program's parsetree. The `method` program part must include the name of its class. The generative program that produces the method is thus parameterized by the `class` program part that corresponds with the class in which the method is located.

Because a generative program can access all dependency and integration relations that originate from its program part, it can deal with the integrative variability to adapt its generated program such that it adheres to the relations. A generative program fetches its required information from other program parts *indirectly.* Instead of directly calling the generative programs that produce the program parts it depends on, a generative program fetches these program parts by referring to the integration and dependency relations that originate from its own part. This means that the generative program retrieves the destination program parts of those relations. Consequently, a generative program automatically has access to all generated program parts that are the destination of a relation that originates from its own generated program part. This indirect calling mechanism is important because a public program part can be involved in many different integration relations, relating to unknown program parts. It allows us to implement a generative program such that it takes all possible integration relations into account for the generation of its program part. An appropriately implemented generative program can thus anticipate and implement the possible generative variabilities caused by integration relations.

Figure 4.8 shows the integrative composition of the tree generator and the traversal generator. It also illustrates the parameterization of the generative programs because of integration relations. Furthermore, even for the generation of private program parts, it is useful to consider integration relations because particular integrative compositions might automatically trigger the inclusion of additional integration relations, even on private program parts. This is explained later on, when we describe *integration relation propagation* in section 4.5.1

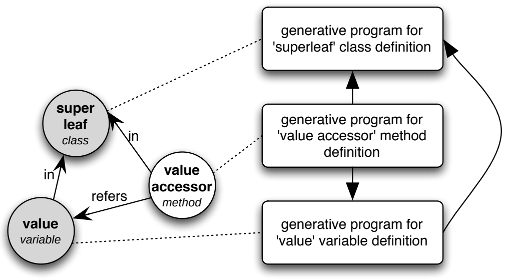

generative program for 'superleaf' class definition
generative program for 'value accessor' method definition
generative program for 'value' variable definition
super leaf
class
in
value accessor
method
in
refers
value
variable

Figure 4.7: Generative programs and their parameterization for parts of the tree generator.

Generation of Listparts

The generative programs that produce listparts instead of simple parts are not really different. Of course, instead of generating one part, they need to produce a list of parts, but the parameterization mechanism works in the same way. Although the integration relations enforce the integration on the individual parts of the list, the generative program itself is parameterized with the entire listpart. In the example depicted in figure 4.8, the generative program that produces the `node traversal` methods listpart is involved in an integration relations with the `node` classes listpart. This generative program is thus parameterized with the entire list of node classes in which its generated traversal methods need to be integrated. The generative program has to generate a list of methods of which each method integrates in an appropriate class. This is enforced by the integration relation.

Generation of Required Parts

In an integrative composition, the required part is integrated with a concrete part. Consequently, all generative programs that are parameterized with dependency relations to the required program part have access to the (integrated) required part. In the example of the traversal generator, depicted in figure 4.8, the generative program that produces the node traversal methods is parameterized with the name of the class it needs to integrate with but also with the name of the method it needs to use to obtain the child nodes. However, the method to access the child nodes is of course generated by the tree generator and not by the traversal generator. The traversal generator declares this dependency by means of a dependency relation to a required part. This `children iteration` required part needs to be integrated with the appropriate `children iterator` part produced by the tree generator. The generative program of the `node traversal` methods list part now has access to the appropriate information in the required part (i.e. the name of the children iterator method). Furthermore, the `node traversal` methods list part is also involved in an integration relationship with the `node` classes list part. Summarized, in this integrative composition, the generative program for the `node traversal` method is parameterized with the `children iterator` method and the `node` class program parts, via the integration relations.

We have previously mentioned that a required program part has no generative program associated with it. This was not entirely true. A required program part does have a generative program part associated with it but this program cannot produce an implementation for the program part without the presence of an integration relation imposed on it. The generative program does not implement the generation of the program part itself. Instead, it implements the retrieval of the other program part with which it is involved in an integration relation. Consequently, the generative program cannot produce a program part without the necessary integration relation that provides access to the other program part. For a required program part in the Smalltalk language, a `unite` integration relation is required. The generative program for the required program part must consequently be implemented to retrieve the other program part that is involved in the `unite` relation. Furthermore, and most importantly, the generative program can enforce particular properties about the program part it is integrated with. If the program part it is integrated with does not adhere to the custom properties imposed by the generative program of the required part, then integration can fail. This is because the generative program of the required program part has control on how the required program part is filled in. To fail the integration, it can violate the integration relation and produce a program part that does not adhere to the integration relation. For example, the required program part `children iteration` of the traversal generator (depicted in figure 4.8) represents the need for a method that iterates over the child nodes. In Smalltalk, such methods are often implemented as methods with a Smalltalk block as argument (e.g. `do:`). Another possibility for iteration of the child nodes is simply a method that returns all child nodes in a list (e.g. `children`). Both these methods have a different signature. Suppose, the traversal generator is written to use an iterator. The generative program of the required program part can enforce that the method integrated with the required part (through a `unite` integration) is a method with the desired signature.

4.4.4 Adaptations for Conflict Resolution

If a composition conflict occurs in the integration of generated programs, it needs to be resolved. In contemporary generative techniques, the developer who composes the generators is required to reconcile the generators through invasive changes in the implementation of the generators or the generated programs. To avoid this difficult and error-prone procedure, our technique enables that the generators adapt their generated program to circumvent the conflict. This adaptation is implemented by means of alternative implementations for the generated program. Each of these alternative implementations circumvents one or more composition conflicts that can occur in an integrative composition. Since the possible composition conflicts are known for each programming language, the developer is able to anticipate the conflicts and consequently specify an alternative implementation.

Alternative Generated Program Parts

A generator can produce alternative implementations for its entire generated program because each generative program, associated with a generated part, can produce one or more alternative implementations. Each alternative implementation that is pro-

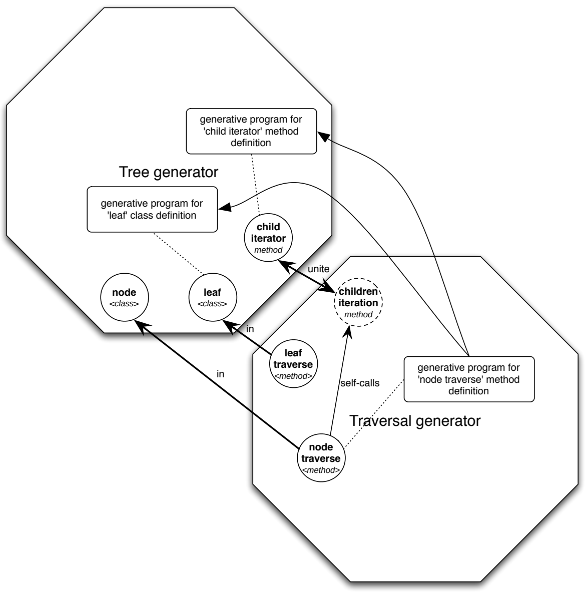

generative program for 'child iterator' method definition
Tree generator
generative program for 'leaf' class definition
child iterator method
node <class>
leaf <class>
unite
children iteration method
in
leaf traverse <method>
in
self-calls
generative program for 'node traverse' method definition
Traversal generator
node traverse <method>

Figure 4.8: Parameterization of the generative programs in the tree-traverse integrative composition.

duced for a particular program part differs from the other alternatives such that it circumvents a particular composition conflict. Since the possible composition conflicts are known for the output language of the generator, the developer can easily specify alternatives for resolvable composition conflicts. The generative programming system will adequately choose the required alternative implementations for each program part such that no composition conflicts occur.

For example, if a generator produces a method named `aux()`, this method can conflict with a method with the same name (produced by another generator). This is because in an integrative composition, this method might be inserted in a class that already contains a method named `aux()`. It can also occur that the method is inserted in a class whose superclass defines a method with same name. This can also lead to strange interferences. To enable an automatic resolution of this conflict, the developer provides an alternative implementation for the program part in which the name of `aux()` method is changed to `auxiliary()` method. The generator can provide multiple alternative names and even provide an unlimited list of renamings for its method. Likewise, if a generator produces a program in the tree language, it should anticipate that the multiplicity of the tree elements can be a conflict if the tree elements are integrated in another tree with a different multiplicity requirement.

The specification of alternative implementations for each generated program part provides a scalable implementation technique to deal with the occurrence of multiple composition conflicts in a single integrated program. In essence, the developer of a program generator only provides alternative implementations for each program part. The complete alternative implementation for the entire generated program is automatically distilled by an appropriate selection of alternative implementations that do not cause composition conflicts and that adhere to all dependency and integration relations. Furthermore, the explicit definition of dependency relations and the parameterization of the generative programs through the dependency relations enables an automatic propagation of alternative implementations, which is described in the following paragraph.

Propagation of Alternatives

In most generated programs, the choice of an alternative implementation for one program part involves that an alternative implementation needs to be chosen for multiple other program parts. This propagation of the selection of alternative implementations happens automatically because the generative programming system enforces the dependency and integration relations. However, this still means that the generative programs that produce the separate program parts need to produce consistent program parts. Fortunately, the developer does not need to implement all possible alternative implementations for each program part explicitly. Instead, the mechanism of parameterization of generative programs by the dependency and integration relations automatically provides each generative program with the alternative implementation of other program parts. In other words, because a generative program is parameterized (via the dependency relations) with other program parts, an alternative implementation for the other program part automatically leads to an alternative implementation for its own generated program part. In essence, the dependency relations ensure that the other generated parts change consistently to produce an entirely correct program.

4.5 The Generative Programming System

We have described how integrative composable program generators are implemented and composed. We will now describe how the generative system that executes the generators is implemented and how it enforces a correct integrative composition. First, we describe how the system automatically deduces additional integration relations to make a particular integrative composition work. Afterwards, the implementation of the conflict detection and integration enforcement mechanism is explained. We also explain how a language definition steers the integrative composition mechanism for the integration of programs implemented in that language.

4.5.1 Integration Propagation

In an integration specification, relations can be drawn between public parts. However, in many cases, an integration relation between public parts can require additional integration relations between other public parts, as well as private parts. However, since an integration relation in an integration specification can only be specified between public parts, such additional integration relations are introduced automatically. For that purpose, the generative system uses a set of propagation rules for integration relations that is supplied for each language. These rules are often necessary to ensure the correctness or even the possibility of integrative composition. In general, a propagation rule ensures that additional integration relations are added because of integration relations defined in an integration specification. A propagation rule is also recursively triggered when an integration relation was added by another propagation rule. The automatic deduction of additional integration relations is necessary because a developer who specifies an integration specification does not know about the internal dependency relations, nor does he know about the private program parts. Consequenty, he is unaware of the need for additional integration relations to make the integrative composition work.

Figure 4.9 illustrates the propagation rules graphically for the Smalltalk language. For example, the top leftmost figure illustrates the propagation rule that an `in` relation induces a `unite` relation if the method or variable in that relation is already involved in an `in` dependency relation, that is internal to the generator. This `in` dependency relation might very well relate this method or variable to a private part produced by the generator. Since no two `in` relations can exist on a method or variable, this integrative composition requires that the parts related to the method or variable via the two `in` relations be integrated by a `unite` relation. The addition of this `unite` integration relation renders the integrative composition correct again.

Integration relation propagation is especially useful if the induced relations operate on private program parts. For example, consider the example of the traversal generator. In addition to the traversal methods, auxiliary methods might be generated that are used by all or some traversal methods. However, it is not particularly appropriate to expose this auxiliary method as a public part in the composition interface. Nevertheless, the auxiliary method also needs to be integrated in the tree program's classes. For this purpose, the traversal generator developer has to define the auxiliary method such that it will automatically be integrated by induced integration relations on the traversal method parts. This is illustrated in figure 4.10. The `in` relation between the `node traverse` methods listpart and `node` classes listpart induces a `unite` relation between the `node` classes listpart and the `traversed node` classes listpart. Subsequently, the `auxiliary method` methods listpart is automatically integrated `in` the `node` classes listpart as well.

4.5.2 Integration and Dependency Enforcement

Integration and dependency relations are enforced by the generative system. For each possible integration or dependency relation in a particular language, the language definition must contain a *constraint* that defines when the two program parts in the relation adhere to the relation. These constraints are used by the generative system to verify if the generated program parts adhere to the integration relation that is applied to them. We have explained how the program generators themselves are influenced by the use of an integration relation in a particular integration specification and how they can adapt the generated program parts to adhere to the integration relation. Since the actual adaptation is done by the program generator itself, the generative system still needs to verify the integration by means of the constraints. This verification also happens in the same way for the dependency relations.

A program generator that anticipates the possible integrative compositions, will produce the correct program parts. However, if the constraint is violated, another alternative implementation for the program part is selected. If no alternative implementation for the program parts exists that adheres to the constraint, then the integrative composition fails. In such a case, it means that the generator is unable to provide an appropriate implementation for the program part that integrates in the other program part. The constraints associated with the integration and dependency

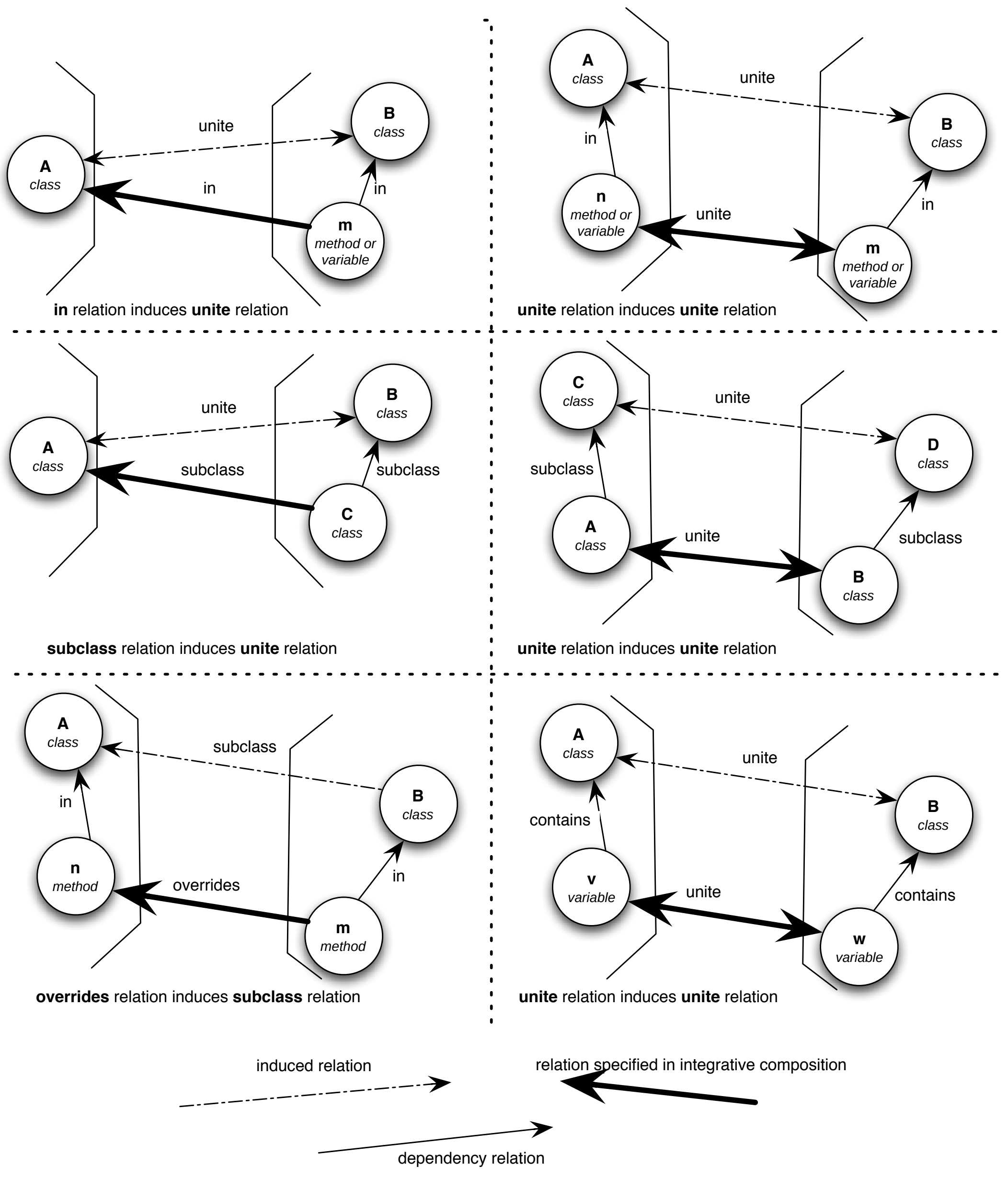


Figure 4.9: Propagation of integration relations in Smalltalk.

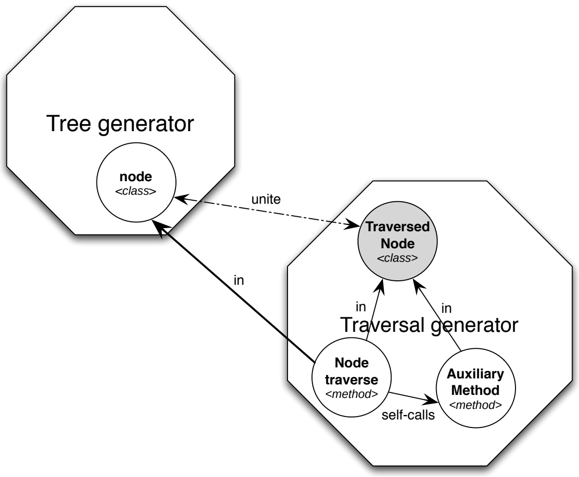


Figure 4.10: Induced integration relation propagates integration.

relations in the Smalltalk language are described next. These descriptions are provided here in natural language, an implementation of these constraints by means of logic metaprogramming is provided in the following chapter.

unite Two program parts that are related via a `unite` integration relation are required to be identical. In essence, they need to represent the same program part in the integrated generated program. The constraint thus enforces that both program parts are entirely identical.

subclass A class `A` that is related to another class `B` via the `subclass` integration relation needs to declare class `B` as its superclass in its representation.

in A method or variable program part that is related to a class program part via the `in` relation needs to declare that it is defined inside that class.

overrides An overrides relation is associated with a constraint that enforces that both methods have an identical signature. The constraint also enforces that the methods are defined in the same inheritance hierarchy.

includeAfter & includeBefore The constraints associated with these integration relations enforce that the method signatures of both methods in the relation are identical. Furthermore, it is also enforced that the method body of the second method in the relation is actually included before or after the rest of the method body of the first method.

contains The contains dependency relation (between a variable and a class) is associated with a constraint that enforces that the type of the variable is the type of the class or a subtype.

refers, calls, self-calls These dependency relations are enforced by a verification if the method actually refers, calls or self calls the appropriate method or variable by its correct name.

4.5.3 Composition Conflict Detection and Resolution Enforcement

Detecting conflicts between generated programs occurs between all generated programs that are integrated. All integrated generated programs in a language are subject to a set of composition constraints that are automatically imposed by the generative system. These constraints are imposed between the program parts of programs implemented in the same language but generated by different generators. These programs can, but must not be involved in an actual integration. A composition conflict is detected as a violation of a composition constraint. We make a distinction between two kinds of composition constraints.

Invalidation Constraints These constraints are associated with the language of the generated programs. They apply to all integrations of programs in that language and are invalidated if the integration produces an invalid program in that language.

Interference Constraints These constraints are also associated with the language of the generated programs. They apply to all integrations and combinations of programs in that language. They are invalidated if two programs interfere with each other and when this interference is not enforced by an integration relation. In other words, these constraints prevent that two programs interfere with each other in undesired ways but they take the presence of integration relations into account because these relations might overrule the interference. In this latter case, the interference is actually a desired interaction that is created by an integration relation.

All composition constraints are specified as binary constraints between the different types of program parts in a language. All these constraints are effectively verified against all possible combinations of appropriate public and private parts of the different generated programs in the same language. In some cases, binary constraints are insufficient and a constraint has to be verified between more than two different parts to detect a composition conflict. This happens if the conflict between two parts depends on some information not included in those parts but in other parts of the generated program. Research in constraint programming has shown that any constraint network with n-ary constraints can be converted to a network with only binary constraints [Bes99]. Consequently, the use of binary constraints does not restrict the detectable composition conflicts. In the implementation of our technique, introduced in the following chapter, we solve this problem because the implementation of a constraint can query the value of other program parts in the system. Consequently, we can express n-ary constraints when necessary.

Table 4.1 describes the composition constraints that are imposed between the generated program parts in the Smalltalk language. The constraints are expressed between the different program part types in Smalltalk: class, method and instance variable definitions. The composition conflicts that are detected by these constraints are purely syntactic. They prevent name clashes between methods, classes and variables, overrides of methods and shadowing of variables. As we have described, some higher-level composition conflicts can be detected in a higher-level language, which will be demonstrated in the following chapter. Also note that the constraint that prevents inadvertent overriding of methods is actually an n-ary constraint as it requires information wether the defining classes of both methods are involved in an inheritance hierarchy. We will show in the implementation with logic metaprogramming how we actually implement these constraints.

Part types	**Smalltalk Composition Constraint**
class - class	Smalltalk classes may not have the same name unless they are related in a `unite` integration relation.
method - method	Smalltalk methods may not have the same name if they are defined in the same class unless they are related in a `unite` relation.
method - method	Smalltalk methods may not override methods in a superclass unless they are related in an `overrides` integration relation. A method may thus not have the same signature as a method defined in a superclass.
instance variable - instance variable	Smalltalk instance variables may not shadow each other. An instance variable may thus not have the same name as an instance variable defined in a superclass.
instance variable - method	Variables defined in a Smalltalk method may not shadow instance variables.

Table 4.1: Smalltalk Composition Constraints

When a composition conflict occurs, the generative system tries to resolve it. This means that the generative system forces the program generators to produce alternative implementations for the generated programs. These alternative implementations are implemented by the program generator itself and are oriented towards circumventing the possible composition conflicts. Composition conflict resolution thus occurs by selecting an alternative implementation for the generated programs that do not conflict. In some cases, this might require another input specification for the program generator. If no alternative implementations can be found that do not conflict, program generation fails. Obviously, not all composition conflicts can be resolved, but there are many cases where it can be resolved.

4.6 Language Definition

Since we have frequently mentioned that particular definitions are part of a language definition, we now summarize what a language definition needs to contain to enable integrations of programs in that language. The language definition serves as a configuration mechanism for the integrative composition technique. A language definition contains:

- The definition of the possible program parts and their representation.
- The definition of integration relations and the constraints that enforce them.
- The implementation of integration propagation rules.
- The definition of dependency relations and the constraints that enforce them.
- The possible composition conflicts and the constraints that detect them.

We described each of these parts of the language definition throughout the chapter where they were needed. In the following chapter, we will explicitly describe them in the definition of a language. Given the knowledge of a language definition, a developer can implement an integrative composable generator that produces programs in that language. More specifically, he can implement the integrative variabilities because he knows about the possible program parts, the integration relations and the compositions conflicts.

4.7 Domain-specific Integrative Composition

The entire mechanism for the implementation of integrative composable program generators is independent of the output language of the program generator. We have shown how a language definition configures the entire generative system for an integrative composition of program generators that produce programs in that language. Although it would be possible to demonstrate the definition of other executable programming languages such as Java, the definition of domain-specific languages represents a more interesting application. By adding the definition of domain-specific languages to our system and by writing program generators that produce programs in these domain-specific languages, we provide the opportunity to accomplish domain-specific integrative compositions. Domain-specific integrative compositions are often more appropriate because they allow us to specify a composition in domain-specific terms instead of the low-level implementation. Furthermore, the detection of composition conflicts at a domain-specific level improves the correctness of the integrated programs because it can be guaranteed that no composition conflicts with respect to that domain occur. Last but not least, an integrative composition at a domain-specific level means that all 'lower level' composition conflicts are also automatically prevented. This is because an integration of domain-specific programs results in a single integrated program that is translated as a whole to the low-level executable language. The low-level implementation is consequently produced by single generator and not by different generators.

4.7.1 Integrative Composition of Composed Generators

Before we consider the actual domain-specific integrative compositions, we need to explain the composition of modularly composed generators. In all previous integrative compositions, we have described the integrative composition of monolithic generators. However, a modularly composed generator can also be involved in an integrative composition. An integrative composition of composed generators is not so much different from an integrative composition of monolithic generators. The integrative composition interface of composed generators is simply the union of the all integrative composition interfaces of their constituents. The difference occurs when the composed generators are implemented as a translation composition of other generators. In such a case, the integrative composition interface of the composed generator does not only expose program parts of the final generated program, but also exposes program parts of the intermediate programs. These intermediate programs are the programs that are 'transferred' between the program generators in a translational composition. Consequently, an integrative composition of composed generators cannot only be specified in terms of the low-level generated programs, but possibly also in terms of the higher-level intermediate programs. Since these intermediate programs are often expressed in a domain-specific language, the integrative composition can be specified at the domain-specific level.

If an integrative composition is specified at the domain-specific level, the domain-specific programs of both generators are integrated into a single program. This integrated program is further translated into the executable language by a single generator. However, each composed generator is built to translate the domain-specific program itself. Therefore, the generative system performs a modification to the composition as illustrated in figure 4.11. On the left hand side of figure 4.11, two composed generators are integrated in terms of the domain-specific programs that describe a tree. Since the domain-specific integration results in an integrated domain-specific program, the generative system merges the composed generators into a single generator that uses only one generator to translate the integrated domain-specific program. This is illustrated on the right hand side of figure 4.11.

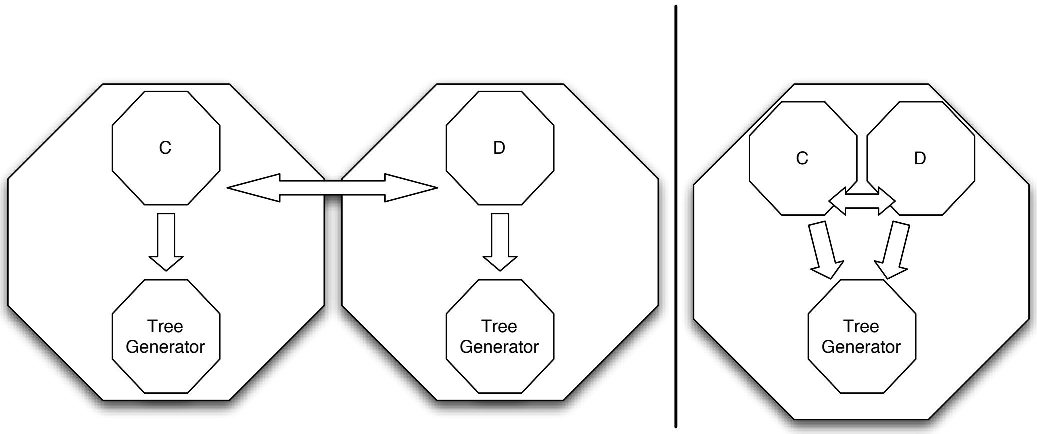


Figure 4.11: Domain-specific Integration results in a merge of the composed generators.

4.7.2 Decomposition in Domain-specific Models

The definition of domain-specific languages occurs naturally in the implementation of modular generators. In the previous chapter, we described that the modularity of a (composed) program generator is best structured such that each program generator in the composition addresses the generation of a single concern. Because each generator is actually a compiler for a domain-specific language, each concern can consequently be described in its own DSL. This is often desirable because it allows that each concern is described in a language that is particularly appropriate for it. This means that in a composed generator, the input specification is either expressed in multiple DSLs or it is internally translated to these different DSLs. In both cases, the modularization according to concerns of the generated program automatically leads to a separation of the input specification in different DSLs, which may or may not be visible by a user of the composed generator.

For example, consider the schematic illustration of the modular composition of program generators in figure 4.12. Each generator in this figure is defined by its input- and output languages. The result of this composition is a generator that produces a program in language $\mathtt{L}_E$ and accepts an input program in languages $\mathtt{L}_A$,$\mathtt{L}_B$ and $\mathtt{L}_D$. In the figure, program `e` is the generated program and programs `a`,`b`,$\mathtt{d}_1$ and $\mathtt{d}_2$ are the input programs. Consider that language $\mathtt{L}_E$ is the executable programming language and all other languages are domain-specific languages that allow us to describe a specific part or concern of the final generated program `e`. We illustrate how in this composition, an integrative composition does not only occur at the level of the program generated in $\mathtt{L}_E$, but there is also an integrative composition at the level of language $\mathtt{L}_C$. This language is situated at a higher-level of abstraction than language $\mathtt{L}_E$ because a program in $\mathtt{L}_C$ is translated to language $\mathtt{L}_E$ by a generator. This domain-specific language $\mathtt{L}_C$ allows us to describe a part of program `e` in domain-specific terms. Consequently, the program `c` *models* a part of program `e` in its own specific language $\mathtt{L}_C$. Therefore, we can also refer to these DSLs as domain-specific *modeling* languages. These DSLs most often describe the final generated program at a higher-level of abstraction and consequently *model* the program.

4.7.3 Example Decomposition

An example of the decomposition in domain-specific models is depicted in figure 4.13. We schematically show the modular structure of the 'parser and traversable parsetree' generator, which is a composition of generators that were introduced in the previous chapter. We can easily identify three different concerns in the final generated program that are internally described in an appropriate DSL:

Parser concern generated by the parser generator and described in the EBNF language.

Tree concern generated by the tree generator and described in the tree language.

Traversal concern generated by the traversal generator and described in the traversal language.

The composed generator accepts a grammar in the grammar language. A grammar description in this grammar language is translated into three programs in different languages: a program in an EBNF language, a program in the traversal language and a program in the tree language. These programs are domain-specific programs that each describe a part of the final generated program. These three programs together

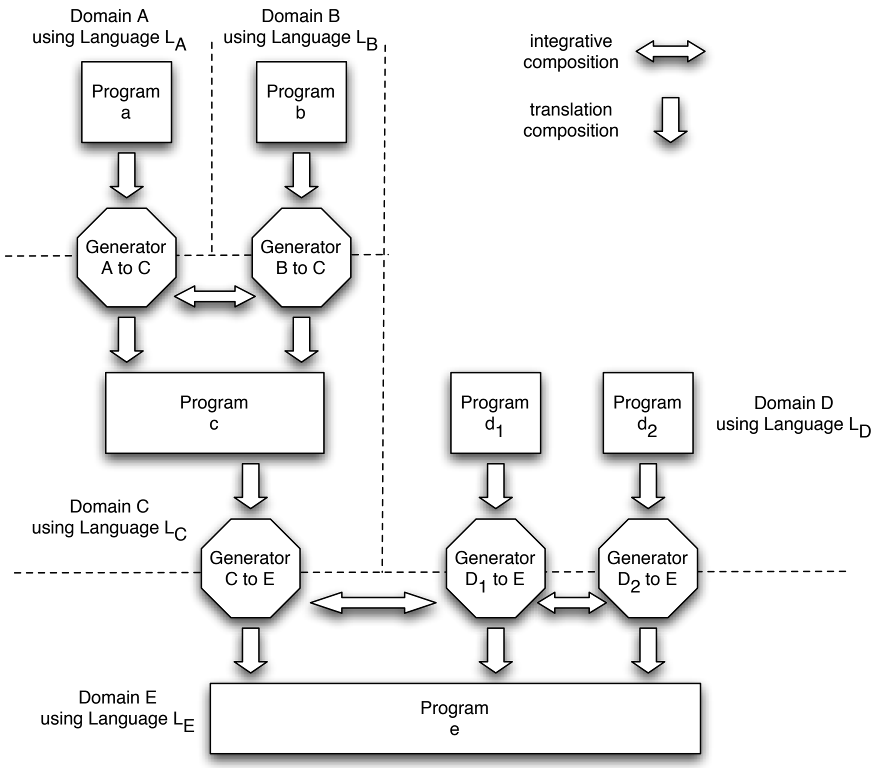


Figure 4.12: Schematic representation of stepwise translation through multiple domain-specific models.

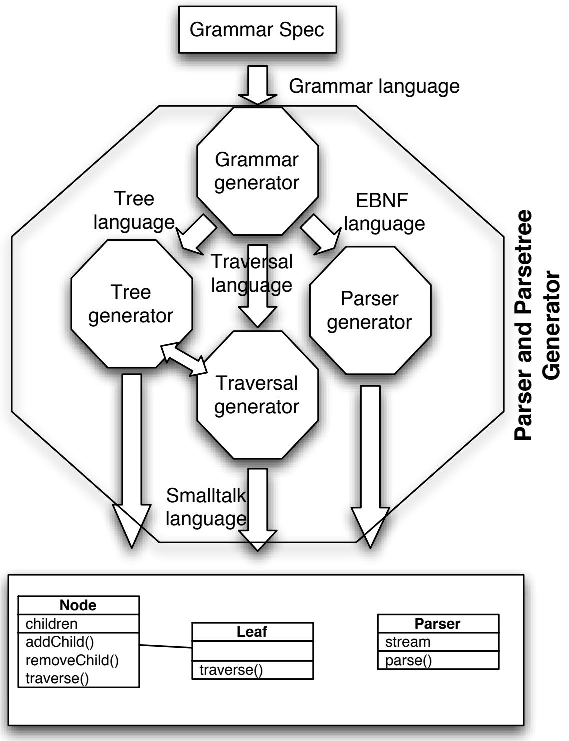


Figure 4.13: Languages and generators in the parser and parsetree generator.

describe the entire generated Smalltalk program. They are further translated into the Smalltalk language by the parser, traversal and tree generators respectively. For the user of the generator, a single language is used to describe the input specification (i.e. the grammar language). The final generated program is produced in Smalltalk and is an integration of the generated programs produced by the parser, traversal and tree generators. The program in the grammar language is a description for the programs in the tree, traversal and EBNF languages because these latter programs are generated based on the grammar program. Thus, there is a fourth part to the modularization that regroups the other three different concerns into a single program, i.e. the grammar program. In the grammar program, some implementation details that need to be included in the tree, traversal and EBNF languages are hidden. Consequently, the grammar language provides some abstraction over the other three languages that are used. In other words, the grammar language describes the final generated program of the 'parser and traversable parsetree' generator at a *higher level of abstraction* than the tree, traversal and EBNF languages.

The stepwise translation of an input program in one DSL, through multiple different DSLs into a final (executable) language is not new. It was first introduced in generative programming by Neigbours in his PhD. dissertation on Draco [Nei80]. As we have discussed in section 2.7.1, chapter 2, the major contribution of Draco is to enable reuse at the domain level instead of at the implementation level. In addition to this major advantage described by Neighbors, the implementation of a generator becomes better structured and easier to maintain because the generation of the separate concerns is not tangled. Model-driven Architectures (MDA) also describe the translation of one model to different specific models [Gro]. In this dissertation, we focus on the application of the integrative composition mechanism at all different levels of abstraction. How higher the level of abstraction, how more domain-specific that the composition can be specified and the more domain-specific that the composition conflicts are that can be detected and possibly resolved.

4.7.4 Example Domain-specific Integrative Composition

The tree generator that was introduced in the previous sections defines a domain-specific language to describe trees. This DSL was briefly described in the previous chapter when we introduced the tree generator. We will now describe the language definition of the `Tree` language and show an example integrative composition of generators that produce a program in this language.

Tree Program Parts

A program in the `Tree` language contains specifications for all nodes and leaves in the tree. We choose to divide the program into the parts that define the separate nodes and the separate leaves. Consequently, the following types of program parts exist in the `Tree` language:

Node This program part represents the definition of a node. Its representation contains the name of the node as well as the names of the nodes and leaves that can be attached as a child of this node in a tree. Furthermore, the definition of a node also includes information on the arity of the node. The arity specifies the number of possible child nodes as a maximum or fixed number.

Leaf The Leaf program part defines a node. A leaf definition consists of a name.

Tree Integration and Dependency Relations

Program generators that produce `Tree` programs can be involved in an integrative composition. The two integration relations that can be defined are:

supportsChild A `supportsChild` integration relation can be imposed between a `node` program part, on the one hand, and a `node` or `leaf` program part, on the other hand. It defines that the first node can now also have the other `node` or `leaf` as a child in a tree.

Unite Is identical to the `unite` integration relation of Smalltalk programs. It can be imposed between `node` or `leaf` program parts.

The `supportsChild` relation is also the only dependency relation that can be used in generators that produce `Tree` programs.

Constraints

The language definition of the `Tree` language also includes constraints that enforce a correct integration of the program parts:

supportsChild In a `supportsChild` integration relation, the first program part must include the name of the other `node` or `leaf` program part in its list of possible child nodes.

Unite In a unite integration relation, the program parts are required to be completely identical.

Tree Composition Conflicts

`Tree` programs that are not integrated cannot conflict. In the event of an identical name for nodes or leaves in different `Tree` programs, a composition conflict will occur at the Smalltalk level. The generator that produces the tree implementation will then propose alternative names for the implementation classes for the conflicting nodes or leaves. However, if tree programs are integrated, all node and leaf names must be unique (as in any `Tree` program). Consequently, the `Tree` language definition contains the implementation of a language-specific invalidation constraint that verifies if all node and leaf names of the (integrated) program are unique.

Integrative Composition of Tree Programs

The integrative composition that is illustrated in figure 4.14 integrates the `Tree` programs into a single `Tree` program using the `supportsChild` and `unite` integration relations. The integrated program expresses the definition of an integrated tree implementation. The resulting generated tree implementation allows to build integrated trees. This integration is not possible at the Smalltalk language level. The integration at the `Tree` language level modifies the entire generated tree implementation. For example, the methods that allow to add a node or leaf to a `NodeD2` node, now also allow to add a `NodeC2` node. An integration at the Smalltalk level cannot express this.

4.8 Development Discussion

The development of integrative composable generators requires careful design and implementation of the program generators. Perhaps the most important part is the language definition that determines the possible integration relations and consequently also the possible integrative compositions of program generators that produce programs in that language. The design and implementation of the integrative composable generators themselves also requires a careful consideration of the possible integrative variabilities that need to be anticipated. In this section, we elaborate on the development methodology of integrative composable generators, as it is proposed in this dissertation. We first discuss the different possible developer roles and discuss the definition of a language and the implementation and integrative composition of program generators afterwards.

4.8.1 Developer Roles

We can distinguish three different developer roles in the implementation of integrative composable generators:

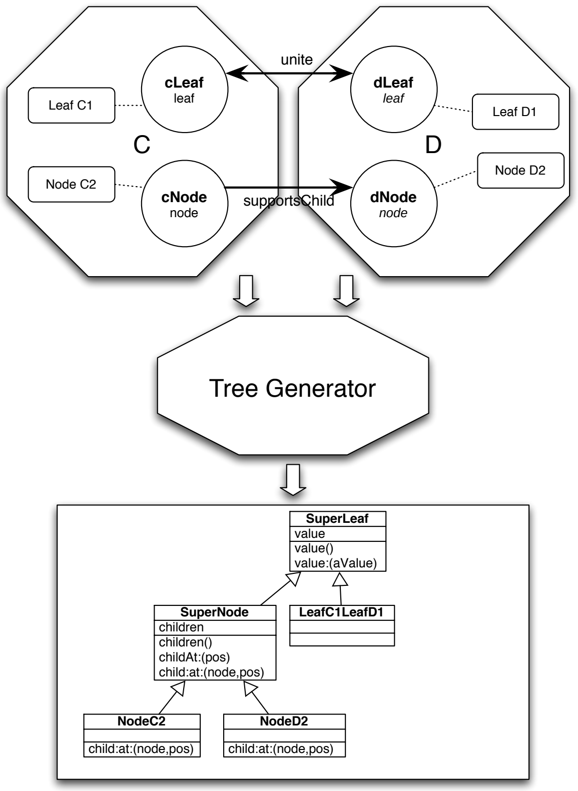


Figure 4.14: Integrative Composition at the Tree language level.

- The developer that implements an entire program generator and defines its input language.
- The developer that builds new generators as an integrative composition of other generators.
- The application developer who uses existing program generators to produce parts of the application.

Although a single person could fulfill multiple developer roles in the development process, it is important to distinguish these roles because each role requires a particular knowledge about the program generator and its generated program. Obviously, the developer of a generator is required to understand the output language(s) of the generator. Using this knowledge, he is able to implement an entire integrative composable program generator. In general, this developer has no knowledge on the implementation of other generators and their generated programs. The second role is the role of the developer that specifies the integrative composition of program generators. This developer also requires knowledge on the input and output languages of the generators in the composition but he does not need any knowledge on their internal implementation. This developer also does not need knowledge on the internal implementation of the generated programs, other than the knowledge that is required to express integrations of these generated programs. The last role is the one of the application developer, who only requires knowledge on the input and output languages of the program generators and the public interface of the generated program.

4.8.2 Language Definition

The definition of a language requires great care because it determines the possible integrative compositions of generators that share this language as output language. To design and define it, the developer needs to consider the possible integrations of programs in that language. First of all, an appropriate set of types of program parts and their representation needs to be determined. In this definition, the developer sets a limit to the kind of program parts that can be manipulated in an integrative composition. Once this set of possible program parts is defined, the developer needs to list the possible composition conflicts that can occur and can be detected between separately generated programs. Obviously, the developer might have to refine the representation of the program parts to ensure that some composition conflicts can be detected.

Next, the developer must define the integration and dependency relations. The choice of program parts determines the possible integration relations. To define the integration relations, the developer needs to distill the useful integrations between the separate program parts. The set of possible program parts, the possible composition conflicts and the possible integration relations then determines to a large extent the required dependency relations for that language. In the definition of the dependency relations, the developer must ensure that all dependencies between the implementations of each separate program part can be expressed. The developer can distill the possible dependency relations by considering what changes to the implementation of one kind of program part require changes to other kinds of program parts, to maintain a consistent generated program. The possible changes to the implementation of a program part are the integrative variabilities. As we mentioned, these variabilities are caused by the possible composition conflicts and integration relations. As a consequence, the developer needs to consider the possible integrative variabilities for each kind of program part and determine how these variabilities can affect the implementation of other kinds of program parts. As we already mentioned, the set of dependency relations frequently overlaps with the dependency relations. However, there can be relations that are specifically designed for integration (e.g. `unite` and `includeBefore`) and some dependency relations are have nothing to do with integration (e.g. `refers`, `calls`).

A last part of the language definition is the definition of the integration relation propagation rules. For this, the language developer needs to consider how an integration relation that is imposed on one program part can require the definition of another integration relation on another program part (that is related via a dependency relation). An integration relation propagation rule is often required to solve impossible configurations of relations on a single program part (e.g. two `in` relations on a `variable` part).

4.8.3 Generator Implementation

The implementation of an integrative composable generator requires that a developer does not only consider the functional variabilities and commonalities of the generated programs. A developer now also needs to consider the integrative variabilities that are caused by the composition conflicts and the integration relations. In his design of the separate program parts, the developer determines the commonalities and variabilities in the generated programs. In essence, the set of program parts and their dependencies is a commonality between all generated programs. The variabilities are limited to the implementation of each separate generated program part. This includes both functional and integrative variabilities. The integrative variabilities can be determined because the developer knows the output language of his generator. The language definition explicitly lists the possible composition conflicts and integration relations that are available. The developer can consequently determine the required integrative variabilities for his generator. The number of variabilities to be considered is in large part determined by the number of public parts exposed in the composition interface. Clearly, a developer who exposes no public parts rules out any possible integrative composition and only needs to consider possible composition conflicts. Once some parts are exposed for integrative composition, the number of integrative variabilities becomes larger. The developer needs to implement the generation of alternative implementations for each generated program part to circumvent possible composition conflicts. Furthermore, each generative program of a program part must consider the possible integration relations imposed on that part and the developer must consequently implement this. The more integrative variabilities that a developer copes with in the implementation of a generator, the more integrative compositions that this generator will support.

4.8.4 Integrative Generator Composition

The specification of an integrative composition is technically quite straightforward. The developer only needs to specify the desired integration relations on the integrative composition interfaces of the generators involved. The integrative compositions are verified for composition conflicts that were identified in the language in which the integrative composition is specified.

4.9 Conclusion

We have described how integrative composable program generators are designed and implemented. In contrast to traditional program generators, such integrative composable generators must anticipate the variabilities that are caused by integrative compositions. The approach we presented allows us to specify integrative compositions through an integrative composition interface by means of integration relations. The possible variabilities caused by composition conflicts and integrative composition itself are captured by the generative programs that can produce multiple possible implementations for a single part of the generated program. The integrative composition mechanism that we presented is language independent and we provided a description of how a language definition can govern the integrative compositions of program generators that share the same output language. This raises opportunities to specify an integrative composition at the domain-specific level.

Chapter 5

Generative Logic Metaprogramming

This chapter presents the realization of the implementation technique for integrative composable generators that was introduced in the previous chapter. We present logic metaprogramming as an appropriate generative programming language for integrative composable generators.

5.1 Introduction

The technique of logic metaprogramming (LMP) [Wuy01, DVMW00, Vol98, MMW02] is a natural implementation technology for integrative composable generators. In the previous chapter, we introduced the overall technique for integrative composable program generators without delving into the details of a particular implementation technology. This is because the proposed technique does not enforce a particular implementation language for an integrative composable generator. In essence, it is possible to use any programming language or even extend existing generative techniques to implement integrative composable generators. However, LMP provides some appropriate linguistic support for the implementation of integrative composable generators. Therefore, we present LMP as an appropriate generative programming language for integrative composable generators.

The use of LMP to generate object-oriented programs was first introduced by De Volder in his PhD. dissertation [Vol98]. In LMP, a program generator is implemented as a logic (meta)program. The logic metaprograms can be used to implement program generators for a program implemented in any kind of language. Most often, programs in an object-oriented programming language have been generated, although HTML webpages and LaTeX source files have been generated using LMP as well [Vol98].

In this dissertation, we have used the Soul logic metaprogramming language [Sou] to implement our generators. Soul is a Prolog [Fla94] derivative, implemented in Smalltalk, that is extended with several reflective and metaprogramming features of which one addresses the manipulation of patterns of (object-oriented) source code. We also chose to use Smalltalk as the object-oriented implementation language of the generated program.

In what follows, we first describe logic metaprogramming by means of the Soul logic metaprogramming language. Afterwards, in section 5.3, we describe that the Generative Logic Metaprogramming system (GLMP) that executes the program generators is a combination of the Soul evaluator and a constraint checker. The implementation of a language definition in GLMP is described in section 5.4. The implementation of the integrative composable generators is explained in section 5.5. Section 5.6 explains the integrative and translation compositions of program generators, followed by sections 5.7 and 5.8 that describe some example integrative compositions at the Smalltalk level and at the domain-specific `Tree` language level, respectively. Finally, a short discussion on the prototype GLMP system is given in section 5.9.

5.2 Logic Metaprogramming

Logic metaprogramming uses a logic programming language at a meta level to manipulate programs in some base language. The technique of LMP itself is not limited to program generation but has been applied in many situations that require metaprogramming in general. LMP has also been used to discover design patterns in object-oriented programs [Wuy98], to enforce programming conventions [MMW02], to co-evolve design and implementation [Wuy01], to describe and deduce software views [MPG03, TBKG04] and aspect-oriented crosscuts [GB03], etc.... In the context of program generation, LMP itself has already been proposed in the context of generative programming [Vol98] and the building of aspect weavers and aspect-specific languages [VD98, BMV02]. In this dissertation, we build upon these last two applications of LMP to support the building of composable program generators.

The logic language that is most often used for logic metaprogramming is a Prolog derivative called Soul [Sou]. Most of the research applications of LMP have been using the object-oriented languages Java and Smalltalk as base languages. In the following sections, we describe how the Soul LMP language differs from standard Prolog and how it is used as a meta language over Smalltalk base programs. We also describe some general ideas of logic programming, but for an in-depth overview on logic programming in Prolog itself, we refer to [Fla94] .

5.2.1 Soul

Soul [Sou] is a logic metalanguage about Smalltalk programs. It was first conceived and implemented by Wuyts in the context of his PhD. dissertation [Wuy01]. Although Soul has a slightly different syntax, it is entirely based on Prolog and can execute normal Prolog programs (converted to Soul syntax). But Soul is much more powerful than a standard Prolog derivative: it has a tight symbiosis with its base language Smalltalk [Wuy01, BGW02, DGJ04]. This means that Soul programs can consist of both Prolog-like logic programs and Smalltalk expressions and that all Smalltalk values can be manipulated in Soul programs. Although the symbiosis itself is of minor importance in this dissertation, its advantage is that generative programs can be implemented in a hybrid programming language. In essence, using Soul, we can use the declarative (logic) and imperative programming styles in the implementation of a single program generator. This allows us to choose the most appropriate programming style to implement parts of a generative program. The declarative (logic) programming style will primarily support the particularities that are associated with integrative composition, while the imperative programming style is more appropriate for algorithmic computations that are often required to generate a program. For more details on the actual symbiosis and its advantages and applications to computational reflection of object-oriented programs, the interested reader is referred to [Wuy01, BGW02, DGJ04].

Soul vs Prolog

Figure 5.1 illustrates the major syntactic differences between Prolog and Soul programs. In essence, Soul logic variables are written with a leading '`?`' instead of a capital letter, as in Prolog. Lists are written between '<' and '>' instead of '[' and ']'. Furthermore, the implication symbol '`:-`' of Prolog is written as '`if`' in Soul. The use of unnamed variables is not illustrated but an unnamed variable is written as `?`. It corresponds to _ in Prolog.

For clarity on terminology we describe how parts of a logic program are denoted in this dissertation. In Prolog or Soul, we do not refer to program statements and expressions, but rather to *logic declarations*. Any logic program consists of multiple logic declarations. A logic declaration can be either a fact or a rule. In the example in figure 5.1, the logic program is defined by one rule and one fact. In both the Prolog and Soul programs, the rule is the first logic declaration and the fact the second declaration. We also denote parts of the logic declarations as *logic terms* and *logic clauses*. A term denotes a part of the logic declaration that is manipulated as data (e.g. the <`?first | ?rest`> lists in figure 5.1). A clause denotes an entire logic declaration that is associated with a truth value. In some Prolog literature, a clause is also defined

```
append(<?first | ?rest>,?aList,<?first | ?restList>) if
   append(?rest,?aList,?restList).
append(<>,?aList,?aList)
```

```
append([First | Rest], Alist,[First | RestList]) :-
   append(Rest,Alist,RestList).
append([],Alist,AList)
```

Figure 5.1: Soul (above) versus Prolog (below) programs.

as a ‘predication’. A clause can be considered as an ‘executable’ part of the logic program (e.g. the `append(?rest,?aList,?restList)` clause in figure 5.1). Terms are also always contained in clauses. Furthermore, we also describe logic programs by the predicates that they define. In the example of figure 5.1, the logic program is a definition for the `append/3` predicate. The predicate of a clause is uniquely determined by a name and *multiplicity*. The multiplicity is a number that denotes the number of arguments associated with that predicate. The `append/3` predicate has a name `append` and a multiplicity `3`.

A single predicate can be defined by multiple logic declarations (i.e. multiple facts and rules). This is an essential part of logic programming. Multiple logic declarations express multiple alternative computations to solve a query. For example, the implementation of the `append/3` predicate in the example above contains two declarations: one fact and one rule. The fact is applicable when the first argument of the predicate is an empty list. The rule is applicable when it is not. The following query will therefore invoke the rule and not the fact. The result returned for `?list` is `<1,2,3,4>`.

```
if append(<1,2>,<3,4>,?list)
```

In this example, the logic declarations are mutually exclusive: either the fact or the rule are executed. However, this does not need to be the case. When more than one logic declaration is applicable, the query can automatically result in *multiple subsequent results*. This is also an essential part of logic programming: a logic query can produce multiple alternative results. For example, the following logic program contains three alternative declarations for the `test/1` predicate. The logic query (also shown below) produces three alternative results for `?x` (i.e. `a`, `b` and `c`).

```
test(a).  test(b).  test(c).
```

```
if test(?x)
```

Quasi-quoted Code

An important feature of Soul in the context of generative programming is the *quasi-quoted code* construct. It provides quasi-quoting facilities in the Soul language and was first introduced in the logic metaprogramming language TyRuBa [Vol], developed by De Volder in the context of his PhD. dissertation [Vol98]. In Soul, quasi-quoted code is written between { and }. It allows us to embed any kind of source code that can be manipulated by the Soul programs. For example, the following Soul clause embeds the source code of a complete Smalltalk method:

```
methodbody({ testmethod
              "this is a demo method source code"
              instancevar := true.
                       ^ self } )
```

Inside the quasi-quoted code, we can still use logic variables. The use of these logic variables enables us to write quasi-quoted source code of which parts still have to be computed during the generation process. In other words, a quasi-quoted code term with logic variables embedded into its quasi-quoted code implements a source code pattern or *template* that is parameterized by the embedded logic variables. The logic variables are instantiated during program evaluation (logic inference process) and they evaluate to their textual representation inside the quoted code.

At the present time, no syntactic limitations are imposed on the contents of a quasi-quotedcode. The Soul language does not limit the use of quasi-quoting to syntactically correct Smalltalk expressions, but also allows any other kind of code such as HTML, XML, Java or even plain strings containing natural language sentences. For that purpose, at the present time, quasi-quoted code is simply considered as a kind of quasi-quoted string. Future extensions to Soul may incorporate a parsing of the code inside a quasi-quotedcode to ensure its syntactic correctness. This would provide additional syntactic correctness checks during code generation.

An example application of a quasi-quotedcode term in generative programs is shown in the following Soul program. It contains a quasi-quotedcode term that implements a template for a Smalltalk instance variable accessor method.

```
accessor(?varName,{ ?varName
              "returns the ?varName variable value"
              ^ ?varName } )
```

The evaluation of the query `if accessor(content,?methodcode)` delivers the accessor method code for the Smalltalk `content` variable:

```
content
  "returns the content variable value"
  ^ content
```

This example illustrates the use of logic variables inside quoted code terms. The presented representation of method code inside a logic clause is different from the representation that is chosen for the implementation of integrative composable generators. This representation is described later on.

5.2.2 Important Predicates

Soul provides a large library of pre-defined standard logic predicates that are frequently used in the implementation of logic programs. We briefly provide an overview of the most important pre-defined predicates that are used in the examples later on.

member/2

The `member/2` predicate verifies or retrieves elements from a list. Its first argument is the element and the second argument is the list. Like any logic predicate, it can be used in multiple ways. For example, it can be used to retrieve all elements from a list:

```
if member(?x,<1,2,3>)
```

The query above results in multiple subsequent results. One result for each element of the list. The member predicate can also be used to verify if an element is contained in the list. The result is either failure or success for that query. For example, the following query succeeds:

```
if member(2,<1,2,3>)
```

findall/3

The `findall/3` predicate is a higher-order predicate, i.e. its second argument is a query. This query is launched and all results for that query are gathered in a list, which is the last argument. The first argument determines which value is put into the list for each result of the query. For example, the following query gathers all elements from the list `<1,2,3>` and gathers them into a new list: `<test(1),test(2),test(3)>`.

```
if findall(test(?x),member(?x,<1,2,3>),?list)
```

5.2.3 Representational Mapping

The use of the logic language Soul at a metalevel to reason about and manipulate Smalltalk base programs requires a mapping of the (object-oriented) base programs to the logic (metalevel) representation. This mapping is called the *representational mapping* and it determines the parts of a program that are reified as separate logic facts [Vol98].

In the specific context of generative programming, this mapping determines the representation of a generated program. Similarly to many other (transformational) generative systems, the internal representation of a generated program in LMP is a parsetree of the program. What is fundamentally different is that this parsetree is not represented as a single logic declaration, but is described by a set of logic declarations that together describe the entire parsetree.

Table 5.1 describes the representational mapping of a subset of the Smalltalk program elements to their internal representation that can be used by the Soul logic metaprograms. This mapping reifies class, variable and method definitions as separate logic declarations:

- A class definition is represented by a logic fact that declares the class name and its superclass name.

- A variable definition is represented by a logic fact that declares the variable name and the class it is defined in. It can optionally define its type, which is useful in integrative compositions. The type is not needed for generation itself because Smalltalk is a dynamically typed language.

- A method definition is represented by a logic fact that declares the method name, the method body and the name of the class it is defined in. The methodbody is represented by a quasi-quotedcode term. Some additional information about the method body is put in the position of the `?info` variable. This additional information is produced by the generator itself and states required information about the method body for verification of the integration and dependency relations. The exact content of this information is discussed later on.

Furthermore, it is important that a representational mapping of a parsetree to its logic representation is a *1-on-1 mapping*. This means that each logic declaration describes exactly one part of the parsetree. Moreover, the inverse mapping of the logic representation to the parsetree must result in the original parsetree. Obviously, any other kind of mapping would lead to strange results. This property of the representational mapping is not verified by the Soul system and consequently needs to be ensured by the developer who defines the representational mapping.

Smalltalk Element	**Logic Representation**
class	class(?name,?superClassName)
variable	var(?className,?name,?Optionaltype)
method	method(?className,?methodName,?methodBody,?info)

Table 5.1: The representational mapping of Smalltalk programs.

5.3 Generative Logic Metaprogramming System

The program generators implemented in the GLMP (Generative Logic Meta Programming) system are executed in a generative system that is based on logic metaprogramming. The Soul logic metaprogramming language is used to implement the program generators and to implement the language definition. The system that executes the program generators is a combination of the Soul interpreter and a constraint checking system. In essence, the definition of a program generator is used by the GLMP system to construct a constraint network. To solve this constraint network, the generative logic metaprograms (included in the generator) are executed. Figure 5.2 illustrates the close interplay of the constraint checking system and the Soul evaluator.

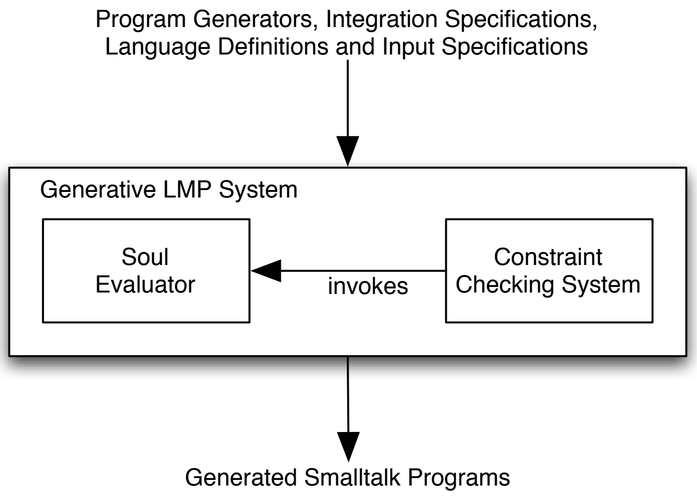

Program Generators, Integration Specifications,
Language Definitions and Input Specifications
Generative LMP System
Soul Evaluator
invokes
Constraint Checking System
Generated Smalltalk Programs

Figure 5.2: The Generative Logic Metaprogramming System.

The Constraint Network

The implementation of a single program generator defines a constraint network. Throughout this dissertation, we have already visualized a program generator as a network of program parts interconnected with relations. This visualization is an exact representation of the constraint network. The network is constructed by the GLMP system itself. The nodes of this network are the separate program parts and the edges of the network are the constraints. A solution of the constraint network is a set of values for each node that satisfies all constraints. In a single program generator, the actual constraints are defined by the dependency relations. The solution of a constraint network defined by a single program generator is thus an entire and correctly generated program. The generated program is represented by a set of program parts. In an integrative composition, the networks of the generators are merged into a single constraint network. The constraints associated with the integration relations implement additional edges that link both networks. Furthermore, the merged network is extended with constraints that prohibit composition conflicts. The GLMP system thus also places edges between the appropriate nodes to implement the composition conflict detection constraints. A correctly integrated program is thus produced by solving the entire merged network.

The constraint checker is the central engine of the GLMP system. It invokes the Soul interpreter to generate the possible program parts (values of network nodes) and selects an appropriate set of alternative implementations for each program part that adhere to the integration and dependency relations. For that purpose, it checks the validity of the relations (implemented as constraints) between the program parts. Since we also implement these constraints in Soul logic metaprograms (as we will describe later on), the constraint checker also invokes the Soul interpreter to verify these relations.

The Generative Logic Metaprogramming Language

The most important use of the logic metaprogramming language Soul is in the implementation of program generators. In other words, Soul is the generative programming language. Consequently, the separate generative programs that produce the separate program parts are called *generative logic metaprograms.* These generative programs can produce multiple possible implementations for a single program part. Nevertheless, we also use Soul to specify the language definition, the relations and all other elements of our approach. We will now describe how a language definition is implemented in GLMP. Afterwards, we explain how the program generators themselves are implemented.

5.4 Language Definition in GLMP

The language definition is included in the implementation of the generator that translates programs in the language. Nevertheless, the language definition and the generator implementation are completely separate. In GLMP, the language definition consists of:

- The representational mapping that determines the program representation and the separate program parts.
- The definition of the possible integration and dependency relations and the constraints that implement their enforcement, implemented using logic metaprograms.
- Integration relation propagation rules, implemented using logic rules.
- Composition conflict detection constraints, implemented using logic metaprograms.

We describe each of these items in detail in the following four subsections.

5.4.1 Representational Mapping of Program Parts

An integrative composable program generator produces a program that is conceptually split into separate program parts. In GLMP, the program generator effectively produces a program in separate parts because it produces a program represented by separate logic facts. The format of these logic facts adheres to the representational mapping of a particular parsetree, as described in section 5.2.3. The representational mapping determines the possible program parts (the possible types) and their entire representation.

The design of a representational mapping requires great care because it is a determining factor for the possible integrative compositions and the handling of integrative variabilities. In essence, the mapping determines the possible kinds of program parts and consequently also determines the kind of program parts that can be exposed through a composition interface. The granularity of the mapping also determines the granularity of the integrative variability that any program generator can offer. For example, the chosen representational mapping of Smalltalk in table 5.1 of section 5.2.3 defines that an entire method is represented by a single logic declaration. This means that the internal implementation of a method is always a hidden part of the generated implementation. Depending on the provided integration relations, this will complicate integrations of generated programs at a sub-method level. Using such a mapping, sub method-level integrations are either impossible or either require more complex integration relations for particular sub-method level integrations. In both cases, it means that generated parts can never contain separate statements. This thus also means that the integrative variability with respect to composition conflicts cannot be expressed at the statement-level. In other words, we can only specify resolutions to composition conflicts by implementing the generation of alternatives for entire method, variable or class definitions. Furthermore, we repeat that the separate program parts defined in a representational mapping need to contain sufficient information to reconstruct the entire program from these separate program parts.

```
class(SuperNode,SuperLeaf).
class(LeafA,SuperLeaf).
class(LeafB,SuperLeaf).
var(SuperNode,children,?).
class(NodeA,SuperNode).
class(NodeB,SuperNode).
method(SuperNode,children,{children ^ children values},<children>).
method(SuperNode,{child:at:},{ child:aChild at:aPos
                              children at:aPos put:aChild },<children>).
method(SuperLeaf,value,{value ^ content},<content>).
class(SuperLeaf,Object).
var(SuperLeaf,content,?).
method(SuperLeaf,{value:},{value:  aValue ^ content := aValue},<content>).
```

Figure 5.3: Sample set of logic facts that represents a part of the tree implementation generated by the tree generator (of figure 4.5).

In this dissertation, we use the representational mapping for Smalltalk programs that was presented in table 5.1. A sample program represented by the logic facts defined in this mapping is shown in figure 5.3. The chosen representational mapping considers entire methods as a single program part. Because the method body is contained in a quotedcode term and remains hidden, the representation of a `method` program part does not only contain the methodbody, the class name and method signature, but also contains an `?info` 'field' that declares properties about the methodbody. These properties declare the use of variables and methods called inside the method body. These properties are produced by the generator that produces the part and they are used by the constraints that implement the verification of dependency and integration relations, which are explained next.

5.4.2 Integration and Dependency Relations

From a technical viewpoint, the implementation and use of integration and dependency relations is identical in GLMP. In the implementation of a program generator or in an integration specification, the dependency and integration relations are specified by means of logic declarations. Therefore, the language definition specifies a logic predicate for each possible integration and dependency relation. The possible Smalltalk integration relations and their corresponding logic declaration are shown in table 5.2. The last column shows the predicate of the constraint that implements its enforcement. The use of the relations in the implementation of a program generator is shown later on. We now focus on the implementation of the enforcement constraints.

Integration Relation	**Logic Declaration**	**ConstraintPredicate**
subclass	subclass(?partA,?partB)	constraintSubclass/4
in	in(?partA,?partB)	constraintVarInClass/4
in	in(?partA,?partB)	constraintMethodInClass/4
overrides	overrides(?partA,?partB)	constraintOverrides/4
unite	unite(?partA,?partB)	constraintUnite/4
includeBefore	includeBefore(?partA,?partB)	constraintIncludeBefore/4
includeAfter	includeAfter(?partA,?partB)	constraintIncludeAfter/4
Dependency Relation	**Logic Declaration**	**ConstraintPredicate**
refers	refers(?partA,?partB)	constraintRefers/4
contains	contains(?partA,?partB)	constraintContains/4
self-calls	selfcalls(?partA,?partB)	constraintSelfcalls/4
calls	calls(?partA,?partB)	constraintCalls/4

Table 5.2: Integration and dependency relations in Smalltalk.

Constraints

The enforcement of the integration and dependency relations is done by constraints implemented with logic metaprograms. The generative system imposes these constraints between the appropriate program parts of one or more program generators and automatically verifies and enforces all the constraints in the execution of a program generator or an integration composition. An invalidated constraint means that the generated program or the integration is incorrect and leads to the selection of alternatives for the program parts. First we describe the general form of how constraints must be declared:

```
?constraintName(?partNameA,?partNameB,?partDescriptionA,?partDescriptionB)
```

Each constraint is defined by its own predicate and must have four arguments. The first two arguments will be bound to the names of the parts that are verified and the last two arguments to the logic representation of these program parts, respectively. Some of the constraints that implement the enforcement of the relations in Smalltalk are shown in figure 5.4. The implementation for the enforcement of the `in`, `refers`, `subclass`,`calls`, `self-calls`, `overrides` and `unite` relations are shown.

The implementation of these constraints extensively uses the unification abilities of the logic inference engine. For example, the `constraintSubclass` logic rule implements the enforcement of the `subclass` relation. This means that the first `class` program part must declare the classname of the second `class` program part as its superclass. This is exactly what that constraint expresses. We also show the use of the additional `?info` 'field' in the `method` program parts. For example, consider the `constraintRefers` constraint, that implements the enforcement of the `refers` relation. There are three logic declarations. The first one verifies if the `method` program part declares that it refers to the `variable` program part using the appropriate name. The method representation must therefore include the name of the variable it refers to in the logic list that is included at the position of the `?info` variable. The list is produced by the program generator itself, during generation of the method. In future implementations, we envision to omit this `?info` 'field' and use Soul's introspective abilities to verify the implementation of the actual method body [Sou]. This is discussed later on in the future work. In this dissertation, all required properties about the method bodies for verification of the constraints is included separately in the `method` program part. This also includes method names for verification of the `calls` and `self-calls` relations. The other declarations of `constraintRefers` implement the enforcement of a `refers` relation if the type of the variable is included and between a method and a class.

A more complex enforcement is required for the `includeAfter` and `includeBefore` relations. The implementation of the constraint for the `includeAfter` integration relation is shown in figure 5.5. First of all, the constraint enforces that both methods declare that they are defined in the same class. Next, this constraint enforces that `method2`, that is combined *after* `method1`, has the same signature as `method2` or is a parameterless method (i.e. a unary message in Smalltalk). We enforce the same signature because the parameters of `method2` need to be mapped onto the parameters of `method1`. Evidently, if `method2` has no parameters, its methodbody can be straightforwardly included. Finally, the constraint needs to verify the correct inclusion of the body of `method2` into the body of `method1`. This requires us to verify the method body of `method1` if the body of `method2` is correctly included (after) the body of `method1`. This verification essentially requires parsetree matching, which is imple-

```
constraintMethodInClass(?m,?c,method(?class,?,?,?),class(?class,?)).
constraintVariableInClass(?v,?c,var(?class,?,?),class(?class,?)).
constraintRefers(?m,?v,method(?,?,?,?info),var(?,?var,?))  if
   member(?var,?info).
constraintRefers(?m,?v,method(?,?,?,?info),var(?,?var,?type)) if
   member(var(?var,?type),?info).
constraintRefers(?m,?c,method(?,?,?,?info),class(?class,?))  if
   member(?class,?info).
constraintSubclass(?c1,?c2,class(?class1,?class2),class(?class2,?)).
constraintCalls(?m1,?m2,method(?,?s1,?,?info), method(?,?s2,?,?))  if
   member(?s2,?info).
constraintSelfcalls(?m1,?m2,method(?,?m1,?,?info),method(?,?m2,?,?))  if
   member(?m2,?info).
constraintOverrides(?m1,?m2,method(?c1,?m,?,?),method(?c2,?m,?,?))  if
   inSameHierarchy(?c1,?c2).
constraintUnite(?,?,?x,?x).
```

Figure 5.4: Implementation of constraints that enforce the Smalltalk dependency relations.

mented by the `includedAfter/2` predicate. In the specific case of an `includeAfter` integration, it is allowed that a custom return statement is included at the end of the method (after the included method body). The implementation of this predicate is included in appendix A.

```
constraintIncludeAfter(?m1,?m2,method(?class,?method1,?body1,?info1),
                                 method(?class,?method2,?body2,?info2)) if
   or(unarySelector(?method2),sameSignature(?method1,?method2)),
   includedAfter(?body1,?body2),
   foreach(member(?x,?info2),member(?x,?info1))
```

Figure 5.5: Implementation of the constraint that enforces the `includeAfter` integration relation.

5.4.3 Integration Relation Propagation

The propagation of integration relations is required because an integration specification might require additional integration relations that can be automatically derived. In GLMP, the propagation of integration relations is implemented using logic rules. Each such logic rule expresses how an integration relation is induced from other integration and dependency relations.

Table 5.3 shows the logic rules that correspond with the integration relation propagation rules for Smalltalk depicted in figure 4.9. In the implementation of these propagation rules, the developer can refer to internal dependency relations in both generators (involved in the composition) via the special `?genA` and `?genB` variables. The generative system executes these rules to determine additional integration relations *before* the generators are executed, i.e. at generator-composition time. During this execution, the variables `?genA` and `?genB` are bound to the generators involved in the composition. Moreover, these rules are executed twice. In the second execution, the binding of the variables `?genA` and `?genB` is swapped because the rules need to be applied symmetrically. To access a rule implemented in a generator, the operator -> is used. For example, the first logic rule in table 5.3 implements the induction of a `unite` integration relation because of a `subclass` integration relation and the presence of a `subclass` dependency relation in one of both generators in the composition. The third logic rule implements the induction of a `unite` integration relation because of a `unite` integration relation and the presence of a `subclass` dependency relation in both generators. A technical detail in these rules is the use of so-called *positive* variables in Soul. These are written with the `+?` notation for variables instead of the normal `?` notation. This is required in the implementation of these rules because their execution can run into a never ending recursion. The solution we adopted is through the use of positive variables because they can only unify with values and not with unbound variables. The positive variables can consequently only unify with a concrete solution and prevent that the execution goes into a deeper recursive process. More information on positive variables can be found in the Soul manual available at [Sou].

5.4.4 Composition Conflicts

Composition conflicts are detected by the invalidation of constraints that are imposed between program parts. In GLMP, the implementation of language-specific invalidation and interference constraints is identical. These constraints are automatically imposed between program parts based on their type. The language definition in GLMP contains the implementation of each of these constraints as logic metaprograms.

First of all, some logic declarations define which constraints need to be imposed

Logic Rule	**Propagation**

`unite(?c,?b) if`
`  subclass(?a,?b),`
`  ?genA->subclass(?a,?c).`
	subclass relation induces **unite** relation because of the `subclass` dependency relation

`unite(?c,?b) if`
`  in(?a,?b),`
`  ?genA->in(?a,?c).`
	in relation induces **unite** relation because of the `in` dependency relation

`unite(?c,?d) if`
`  unite(+?a,+?b),`
`  ?genA->in(?a,?c),`
`  ?genB->in(?b,?d).`
	unite relation induces **unite** relation because of the `in` dependency relations

`unite(?c,?d) if`
`  unite(+?a,+?b),`
`  ?genA->subclass(?a,?c),`
`  ?genB->subclass(?b,?d)`
	unite relation induces **unite** relation because of the `subclass` dependency relations

`unite(?c,?d) if`
`  unite(+?a,+?b),`
`  ?genA->contains(?a,?c),`
`  ?genB->contains(?b,?d)`
	unite relation induces **unite** relation because of the `contains` dependency relations

`subclass(?c,?d) if`
`  overrides(?a,?b),`
`  ?genA->in(?a,?c),`
`  ?genB->in(?b,?d),`
`  not(inSameHierarchy(?c,?d))`
	overrides relation induces **subclass** relation because of the `in` dependency relations and because the classes are not defined in the same hierarchy

Table 5.3: Logic rules that implement integration relation propagation for Smalltalk.

between which types of program parts. This association is implemented using the `constraintDef` predicate.The Smalltalk language definition contains the following declarations:

```
constraintDef(method,method,<methodOverridesConstraint,methodUnitesConstraint>).
constraintDef(var,var,<varUnitesConstraint,varShadowsConstraint>).
constraintDef(class,class,<classConstraint>).
```

The first declaration states that all `method` program parts produced by different generators are subjected to the `methodOverridesConstraint` and `methodUnitesConstraint` constraints. These constraints ensure that no two methods defined in the same classes or in the same hierarchy and produced by different generators have the same name, unless they are involved in an `overrides` or `unite` integration relation. We can also see that the `varShadowsConstraint` is imposed between all program parts of the type `var`, produced by different generators.

Implementation of Composition Conflict Detection Constraints

The implementation of the constraints themselves does not only need to check the implementation of the different program parts but also needs to check the occurrence of integration relations between these program parts. This is because a composition conflict is never a conflict when it was enforced through an integration relation. For example, the `overrides` integration relation invalidates the 'inadvertent method overriding' composition conflict. Therefore, the implementation of a composition conflict detection constraint is often expressed with multiple alternative logic declarations. One that is applicable in the absence of integration relations and one for each possible integration relation that can be imposed between the two parts. The developer that implements the constraints must make sure that in the case of a composition conflict, all alternatives fail. We illustrate this with an example constraint definition. Additional composition conflict detection constraints are included in appendix A.

The following logic rules declare two constraints that are applicable between all instance variable definitions of different generated programs. The first constraint is defined by the predicate `varUnitesConstraint` and checks if no two instance variables are defined on the same class with the same name. This is a composition conflict that can occur if two class definitions are integrated by a `unite` relation. However, in case of such a `unite` relation between the variable definitions themselves, the composition conflict should not occur. Evidently, this means that both variables are meant to be the same in the integrated programs and consequently, both program generators produce an identical variable definition, which in that case, is no composition conflict.

The second constraint is defined by the `varShadowsConstraint` predicate. It prevents that variables defined in subclasses shadow variables defined in a superclass.

```
varUnitesConstraint(?nameA,?nameB,var(?classA,?varA,?typeA),
                              var(?classB,?varB,?typeB)) if
   not(or(unite(?nameA,?nameB),unite(?nameB,?nameA))),
   not(and(equals(?classA,?classB),equals(?varA,?varB))).
varUnitesConstraint(?nameA,?nameB,?,?)  if
   or(unite(?nameA,?nameB),unite(?nameB,?nameA)).

varShadowsConstraint(?nameA,?nameB,var(?classA,?varA,?),
                              var(?classB,?varB,?))  if
   not(and(inSameHierarchy(?classA,?classB),equals(?varA,?varB)))
```

5.4.5 Additional Logic Metaprograms

To complete the implementation of the language definition, we need to mention that a language definition can also contain the implementation of a set of logic metaprograms. This set of logic metaprograms can be used by all generators that have this language as their output language. The additional logic metaprograms are often useful because they implement a frequently needed computation to generate programs in that language.

```
includeAfter(?partA,?selectorA,?argumentsA,?includeBody,?includeInfo) if
   includeAfter(?partA,?partB,method(?,?selectorB,?body,?includeInfo)),
   or(unarySelector(?selectorB),sameSignature(?selectorA,?selectorB)),
   convertAndStripHeader(?body,?argumentsA,?includeBody).
includeAfter(?partA,?selectorA,{ },<>) if
   not(includeAfter(?partA,?partB,?))
```

Figure 5.6: Special `includeAfter/4` predicate supplied with the Smalltalk language.

For example, the `includeAfter/4` predicate is implemented by such an additional logic metaprogram in the Smalltalk language definition. Its implementation is shown in figure 5.6. This predicate always returns a result. If an `includeAfter` relation was declared with `?partA` as origin, it returns the method body of the method in the destination of the `includeAfter` relation. If no `includeAfter` relation was declared, it simply returns an empty method body. The implementation of the predicate also verifies if the methods can be related in an `includeAfter` relation. Therefore, it

verifies the signatures of the methods for compatibility and it automatically substitutes the method arguments in the method bodies such that they can form a single method body.
This concludes the language definition in GLMP.

5.5 Integrative Composable Generators in GLMP

The implementation of a program generator consists of three parts:

- Dependency relation declarations
- Generative logic metaprograms
- Additional logic metaprograms
- Implementation of generator-specific dependency relations

Except for generator-specific dependency relations, we discuss each of these parts in detail in the following subsections. The generator-specific dependency relations are defined in the same way that integration and dependency relations are specified and implemented in the language definition. They can be included in the implementation of a single generator to express a particular dependency relation required in its implementation.

5.5.1 Dependency Relation Declarations

The dependency relations between program parts are defined using logic facts. The format of these logic facts is defined in the language definition of the output language of the generator. For example, the logic facts that declare the dependency relations used in the implementation of the tree generator are shown in table 5.4. This set of logic declarations implements the relations shown in figure 4.5.

5.5.2 Generative Logic Metaprograms

The core functionality of a program generator, i.e. the generation of the actual program parts, is implemented by generative logic metaprograms. Each separate program part (and its associated generative logic metaprogram) is defined by a single logic predicate. To illustrate this, we show the implementation of the tree generator in table 5.5, where each row corresponds to a generated part of the tree program. The first and second column show the generated part's name and its corresponding predicate in the implementation of the generator. The last column shows the generative logic metaprogram that implements this predicate and thus generates the part.

Implementation Dependency	**Relation Declaration**
Super Node *subclass of* Super Leaf	`subclass(superNode,superLeaf)`
Node *subclass of* Super Node	`subclass(node,superNode)`
Leaf *subclass of* Super Leaf	`subclass(leaf,superLeaf)`
Children *defined in* Super Node	`in(children,superNode)`
Value *defined in* Super Leaf	`in(value,superLeaf)`
General Setter *defined in* Super Node	`in(genSetter,superNode)`
Child Setter *defined in* Node	`in(childSetter,node)`
Child Accessor *defined in* Super Node	`in(childAccessor,superNode)`
Value Accessor *defined in* Super Leaf	`in(valueAccessor,superLeaf)`
Value Setter *defined in* Super Leaf	`in(valueSetter,superLeaf)`
General Setter *refers to* Children	`refers(genSetter,children)`
Child Accessor *refers to* Children	`refers(childAccessor,children)`
Value Accessor *refers to* Children	`refers(valueAccessor,children)`
Value Setter *refers to* Children	`refers(valueSetter,children)`
Children Iterator *refers to* Children	`refers(childrenIterator,children)`
Children Iterator *defined in* SuperNode	`in(childrenIterator,superNode)`

Table 5.4: Dependency relations in the tree generator.

We explain the implementation of the generative logic metaprograms in the following subsections, with a special focus on the handling of the integrative variabilities.

Implementation of Conflict Resolution

To resolve possible composition conflicts, each generative logic metaprogram must generate multiple versions for each generated part. Logic programming has the interesting property that a single logic program can produce multiple subsequent results. Each of these results corresponds to an alternative implementation of the generated program part. The GLMP system automatically selects the appropriate alternative that does not result in a composition conflict (and that adheres to all relations).

We can easily implement these alternative results of a logic program by writing multiple logic declarations that implement the same logic predicate. It means that each generative logic metaprogram can be defined by a set of declarations that each define an alternative implementation for a generated part. This native linguistic feature in logic programming makes the LMP approach particularly interesting for specifying the generation of alternative implementations.

For example, in the implementation of the tree generator, shown in table 5.5, the logic metaprogram that produces the `Value` part contains two declarations that each implement the generation of a different implementation for the `Value` part. In one implementation, the variable is named `content` while in the other implementation, the variable is named `alternateContent`. This alternative implementation resolves the composition conflict where another variable with the same name is integrated in the same class. Quite evidently, more than one alternative implementation can be implemented and a logic metaprogram can even implement a never ending list of alternative implementations. Obviously, such a never ending list of alternatives might cause that the generation is a non-halting process. However, a simple solution is that the generative system sets a maximum number of alternatives that are considered. This is somewhat similar to what is done in some interpreters for logic programs as well, by specifying a so-called 'cut-off' depth for the recursion on the runtime stack [Fla94].

Parameterization by Relations

To anticipate the possible integration relations and to produce a correct program part that adheres to these relations, the generative logic metaprograms also need to be parameterized by the possible integration and dependency relations that may operate on their generated part. This involves the generative logic metaprogram calling the predicate that defines the relation, but with multiplicity three instead of two. The additional argument contains the program part in the destination of the relation. The implementation of this predicate is automatically provided by the generative system for each relation predicate. For example, for the Smalltalk language, the generative system automatically provides the predicates `unite/3`, `in/3`, etc.... These predicates need to be called by the generative programs to verify the presence of a relation and to fetch the program part in the destination of that relation.

For example, the generative logic metaprogram for the `SuperLeaf` part, shown in table 5.5 calls the `subclass/3` predicate. The first two arguments correspond to the source and destination part names of the relation. The last argument contains the logic representation of the destination program part. This call explicitly parameterizes this generative program with the `subclass` relation between its own part (the first argument in the call: `SuperLeaf`) and any other part it is integrated with through a `subclass` integration relation. In this specific example, only the second logic declaration of the `SuperLeaf` part is parameterized with this `subclass` relation. The first declaration states that the generated `SuperLeaf` class part is a subclass of `Object`. This means that the generative program normally produces its class as a direct subclass of `Object`. However, an alternative implementation can be produced (by the second logic declaration) if the `SuperLeaf` part is involved in a `subclass` integration relation. In that case, the generated `SuperLeaf` program part will be a subclass of the `class` program part that is the destination of the `subclass` integration relation. The alternative implementation is automatically selected because the constraint checker enforces that the `SuperLeaf` program part adheres to the integration relations imposed on it.

The required adaptations for integration of a program part are thus also implemented through the generation of alternative implementations for that program part. Multiple logic declarations in a single generative logic metaprogram allow us to deal with the presence and absence of integration relations. We can implement a generative logic metaprogram as a set of logic declarations of which different declarations anticipate different integration relations. Each separate logic declaration produces an alternative implementation of the program part that adheres to particular integration relations.

Dealing with Circular Dependencies

In the implementation of a generator, it is possible that a circular dependency is created between program parts. Consider for example, the circular dependency shown in figure 5.7. The method in program part `methodA` calls the method in part `methodB` and vice-versa. The generation of either one of these program parts thus requires the generation of the other program, which is a circular dependency. In GLMP, the system can consequently run into an infinite loop. Nevertheless, this can be easily solved by the developer of the generator through the definition of partially generated program parts. For example, consider the generative logic metaprograms that produce the `methodA` and `methodB` program parts in figure 5.8. Both generative programs implement an alternative declaration that produces a partial implementation of the method. This partial implementation only contains the information required by the other generative program. In this example, this is the method name (e.g. `selectorA` and `selectorB`).

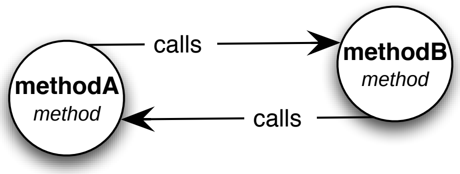


Figure 5.7: Circular dependency between program parts.

```
methodA(method(?class,selectorA,{},<>)).
methodA(method(?class,selectorA,{selectorA self ?selectorB },<?selectorB>)) if
   in(methodA,?,class(?class,?)),
   calls(methodA,methodB,method(?,?selectorB,?,?)).

methodB(method(?class,selectorB,{},<>)).
methodA(method(?class,selectorB,{selectorB self ?selectorA },<?selectorB>)) if
   in(methodB,?,class(?class,?)),
   calls(methodB,methodA,method(?,?selectorB,?,?)).
```

Figure 5.8: Partially generated program parts to solve circular dependencies.

5.5.3 Additional Logic Metaprograms

Besides the logic metaprograms that are directly associated with the generation of a program part, there are other logic metaprograms defined in the implementation of a program generator. These other logic metaprograms can be seen as auxiliary programs that are called by one or more generative logic metaprograms that directly generate a part. This allows the developer to factor out some common behaviour or a complex procedure in a separate logic program.

The generative programs of the tree generator (shown in table 5.5) also call a number of additional logic metaprograms that implement a part of the generation process. Some example predicates that are called are `makeMethodHeader/3`, `valAccessorName/1`, `makeChildSetters`, etc.... The logic metaprograms that implements these predicates implement a part of the generation process. Some of these logic metaprograms implement the generation of code that can be used in the implementation of many program generators. For example, the `makeMethodHeader` program assembles a Smalltalk selector (methodname) and arguments into a Smalltalk method header. Such a header must be included in the generation of each Smalltalk method. Other logic metaprograms are very particular to the implementation of the tree generator. For example, the `makeChildSetters` program implements the generation of the list of methods that implement the addition of children to a node. This method needs to include checks that only allow the addition of the right children to the node.

5.6 Generator Composition

5.6.1 Integrative Composition

The definition of an integrative composition boils down to declaring a set of integration relations between public parts produced by different generators. Their declaration in GLMP is identical to the declaration of relations used in the implementation of a program generator, i.e. using logic declarations. Table 5.6 shows the declarations for an integrative composition of the traversal and tree generators as shown in figure 4.8. Mind that in our current GLMP implementation, the names of all program parts of all generators must be unique. This is a technical detail that could easily be omitted in future implementations.

5.6.2 Translation Composition

In a translation composition of generators, one generator needs to retrieve its input program from another generator. In our approach, a generator makes no distinction between an input specification that was written by a developer or one that was generated by another generator. In other words, all input specifications are assumed to be produced by another generator. In the development of a generator, we prepare for translation composition through the parameterization of the logic metaprograms. However, in this case, the logic metaprogram is not parameterized by another logic metaprogram but by the input program. Each logic metaprogram in the implementation of a generator can retrieve the appropriate parts of the input program through the pre-defined prediate `retrieveInput/2`. This predicate allows to retrieve all program parts of a certain type in the generator that provides the input program. For example,

Part	Predicate	Declarations

Super Leaf	`superLeaf/1`	`superLeaf(class(SuperLeaf,Object))`
		`superLeaf(class(SuperLeaf,?superclass)) if`
		`  subclass(superLeaf,?externalPart,class(?superclass,?))`

Super Node	`superNode/1`	`superNode(class(SuperNode,?superLeaf)) if`
		`  subclass(superNode,superLeaf,class(?superLeaf,?))`

General Setter	`genSetter/1`	`genSetter(method(?superNode,?childSetter,{?header ?childrenVar at: pos put: child},?childrenVar)) if`
		`  refers(genSetter,children,var(?superNode,?childrenVar,?)),`
		`  in(genSetter,superNode,class(?superNode,?)),`
		`  childSetterName(?childSetter)`
		`  makeMethodHeader(?childSetter,<aChild,aPos>,?header)`

Child Setter	`childSetter/1`	`childSetter(?methods) if`
		`  refers(childSetter,children,var(?,?childVar,?)),`
		`  in(childSetter,nodes,?nodeClasses),`
		`  overrides(childSetter,genChildSetter,method(?,?selector,?,?)),`
		`  retrieveInput(treespec,node,?nodeSpecs),`
		`  makeChildSetters(?childVar,?nodeClasses,?selector,?nodeSpecs,?methods)`

Child Accessor	`childAccessor/1`	`childAccessor(method(?superNode,?childAccessor,{?childAccessor ^?childrenVar},<?childrenVar>)) if`
		`  in(childAccessor,superNode,class(?superNode,?)),`
		`  refers(childAccessor,children,var(?superNode,?childrenVar,?)),`
		`  childAccessorName(?childAccessor)`

Children Iterator	`childrenIterator/1`	`childrenIterator(method(?superNode,{do:},{ do: aBlock ?childrenVar do: aBlock. },<?childenVar>) if`
		`  in(childrenIterator,superNode,class(?superNode,?)),`
		`  refers(childrenIterator,children,var(?superNode,?childrenVar,?))`

Children	`children/1`	`children(var(?superNode,children,?)) if`
		`  in(children,superNode,class(?superNode,?))`

Value	`value/1`	`value(var(?superLeaf,content,?)) if`
		`  in(value,superLeaf,class(?superLeaf,?))`
		`value(var(?superLeaf,alternateContent,?)) if`
		`  in(value,superLeaf,class(?superLeaf,?))`

Value Accessor	`valAccessor/1`	`valAccessor(method(?superLeaf,?valueAccessor,{?valueAccessor ^?valueVariable},<?valueVariable>)) if`
		`  in(valAccessor,superLeaf,class(?superLeaf,?)),`
		`  refers(valAccesoor,value,var(?superLeaf,?valueVariable,?)),`
		`  valAccessorName(?valueAccessor)`

Value Setter	`valSetter/1`	`valSetter(method(?superLeaf,?valueSetter,{?valueSetter ^?valueVariable := newValue},<?valueVariable>)) if`
		`  in(valSetter,superLeaf,class(?superLeaf,?)),`
		`  refers(valSetter,value,var(?superLeaf,?valueVariable,?)),`
		`  valMutatorName(?valueSetter)`

Node	`node/1`	`node(?classList) if`
		`  subclass(node,superNode,class(?superNode,?)),`
		`  findall(class(?name,?superNode),nodeName(?name),?classList))`

Leaf	`leaf/1`	`leaf(?classList) if`
		`  subclass(leaf,superLeaf,class(?superLeaf,?)),`
		`  findall(class(?name,?superLeaf),leafName(?name),?classList))`

Table 5.5: Main implementation of the tree generator.

Integration Relation	**Implementation**
Leaf Traverse *in* Leaf	`in(leafTraverse,leaf)`
Node Traverse *in* Node	`in(nodeTraverse,node)`
Children *unite* Child Iterator	`unite(children,childIterator)`

Table 5.6: Implementation of the integrative composition of the tree and traversal generators.

in the implementation of the tree generator in table 5.5, the logic metaprograms that define the `Leaf` and `Node` parts access the input program through the `nodeName` and `leafName` predicates. These predicates are implemented as separate logic metaprograms that retrieve the node and leaf names through the `retrieveInput/2` predicate:

```
leafName(?name) if
   retrieveInput(leaf,?leafSpecs),
   member(leaf(?name),?leafSpecs).
```

The `leafName/1` predicate is implemented as a call to the `retrieveInput/2` predicate. The first argument of this predicate is the program part type that needs to be gathered from the input program. The second argument contains the list of program parts of that type in the input program. Mind that if the input program is a program produced by another generator, there can be multiple possible implementations of that input program. The generative programming system ensures that all generative logic metaprograms of one generator consistently use the same input program (in the generation of one output program). In a translation composition, a generator thus fetches a possible input program and uses it to generate its own program (i.e. it translates the input program to the output program). In case that this input program cannot lead to a correct generated program (that, for example, integrates with another program) because of composition conflicts, then another input program is (automatically) chosen if one is available.

5.7 Integrative Composition Examples

We will now focus on the implementation of some interesting examples that were mentioned in the previous chapters.

5.7.1 Invasive Integration of Variable Program Parts

An interesting invasive integration is the `unite` integration applied to `variable` program parts. The `unite` integration enforces both variables to be identical. For the purpose of conflict detection in integrative compositions, we have added an optional type declaration to the representation of `variable` program parts. In a `unite` integration relation, this means that these types must also be identical. Since the rest of the generated program often heavily relies on the type of the variable, the integration of variables often requires to adapt an entire generated program.

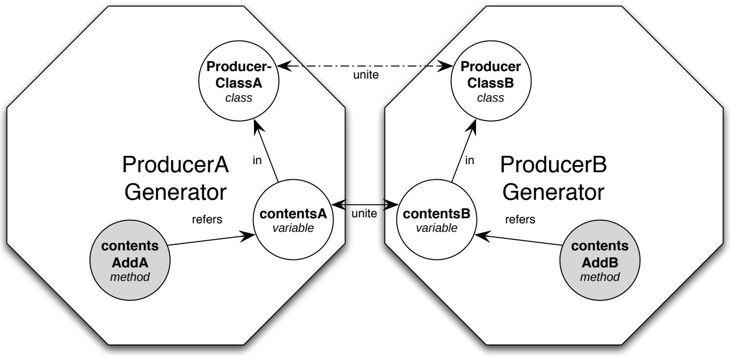


Figure 5.9: Integrative Composition for the integration of variables.

Consider the integrative composition of two generators (`ProducerA` and `ProducerB`) that each generate a class that contains some code and a variable. This example is actually a simplification of the example integrative composition of the graph generator and observer-observable generator, introduced in chapter 3. Here we only focus on the required adaptations for the integration of the variables produced by both generators.

The integrative composition that is shown in figure 5.9 integrates both variables (`ContentsA` and `ContentsB`) through a `unite` integration relation. This integration automatically triggers the propagation mechanism and induces a `unite` relation between their classes. `ProducerA` prefers to generate code that uses a `Set` as type of the variable because it wants to rely on the implementation of `Set` to avoid duplicate elements. Consequently, the `ProducerA` generator produces a first alternative implementation where the `contents` variable is typed as a `Set`. However, `ProducerB` can only work with a variable typed as `OrderedCollection`. Since an `Orderedcollection` can contain duplicate elements, an integration of these variables would lead to a broken functionality of the program produced by `ProducerA`. This interference is detected because the variables do not `unite`, i.e. their type is different. Consequently, the integrative composition would fail. Fortunately, the `ProducerA` generator can also produce an alternative generated program with a variable typed as `OrderedCollection`. Therefore, the integrative composition will work because the integrative composition forces the selection of the alternative implementation. However, the entire generated program of `ProducerA` needs to be adapted such that the generated program of `ProducerA` itself verifies for duplicate elements in the collection. This adaptation is shown in the generative programs of the `contentsA` and `contentsAddA` program parts in figure 5.10:

- The generative program for the `contentsA` part can produce a `variable` program part that is typed as a `Set` or an `OrderedCollection`.

- The generative program for the `contentsAddA method` part can produce a method that works with a `Set` and a method that works with an `OrderedCollection`. The implementation that works with a variable typed as an `OrderedCollection` checks if an element is not included in the collection before it adds it. The implementation that works with a variable types as a `Set` simply adds it. It consequently relies on the implementation of the `Set` to omit duplicates. The selection of the appropriate alternative implementation is (automatically) enforced through the `refers` dependency relation (shown in figure 5.9).

```
contentsA(var(?class,contents,Set)) if
   in(contentsA,producerA,class(?class,?)).
contentsA(var(?class,contents,OrderedCollection)) if
   in(contentsA,producerA,class(?class,?)).

contentsAdderA(method(?class,{add:},{add:el ?var add:el},<var(?var,Set)>)) if
   in(contentsAdderA,producerA,class(?class,?)),
   refers(contentsAdderA,contentsA,var(?class,?var,Set)).
contentsAdderA(method(?class,{add:},{add:el (?var includes:el)
                                       ifFalse:[?var add:el]},
                                  <var(?var,OrderedCollection)>)) if
   in(contentsAdderA,producerA,class(?class,?)),
   refers(contentsAdderA,contentsA,var(?class,?var,OrderedCollection)).
```

Figure 5.10: Generative programs of the ProducerA generator.

5.7.2 Integration of Required Program Parts

Required program parts need to be integrated with appropriate program parts, produced by another generator. The type of the required program part already limits the possible integrations but the generative program that is associated with the required program part can limit the possible integrations even further. The generative program of the required program part must produce a program part that adheres to the integration relation. Since it does not implement the generation itself, it merely 'copies' the program part that is integrated with the required part via a `unite` integration relation. A simple generative program for a required `class` part thus looks like:

```
requiredPart(?class) if unite(requiredPart,?,?class).
```

However, the generative program of the required part can be implemented such that the other program part is not simply copied into the required part, thereby violating the integration relation. As a consequence, the integrative composition fails. Using this mechanism, a program generator can enforce particular properties on the program parts that are integrated with the required program parts. Of course, this mechanism is not limited to generative programs associated with required program parts but it is particularly useful to them.

An example integrative composition where this is useful is shown in figure 5.11. This example was already introduced in the previous chapter and is about the integrative composition of the traversal generator with the tree generator. The traversal generator contains a required part (`ChildrenIteration`) that needs to be filled in with the method that provides access to the child nodes in a node. This access can be either provided through a Smalltalk iterator method (e.g. `do:`) or via a method that returns all child nodes in a list (e.g. `children`). Both kinds of methods impose a different way of iterating over the children. The iterator method needs to be called with a Smalltalk block argument that implements an action for each node. The second kind of method simply returns all child nodes and leaves the iteration up to the caller of the method. Based on the differences in method signatures, the traversal generator can impose the use of a Smalltalk iterator. Therefore, the generative program of the `ChildrenIteration` required part is as follows:

```
childrenIteration(method(?class,?selector,?body,?info) if
   unite(childrenIteration,?aPart,method(?class,?selector,?body,?info)),
   singleKeyword(?selector).
```

The above generative program retrieves the `method` program part that is integrated with it through a `unite` integration relation and imposes that the method has only

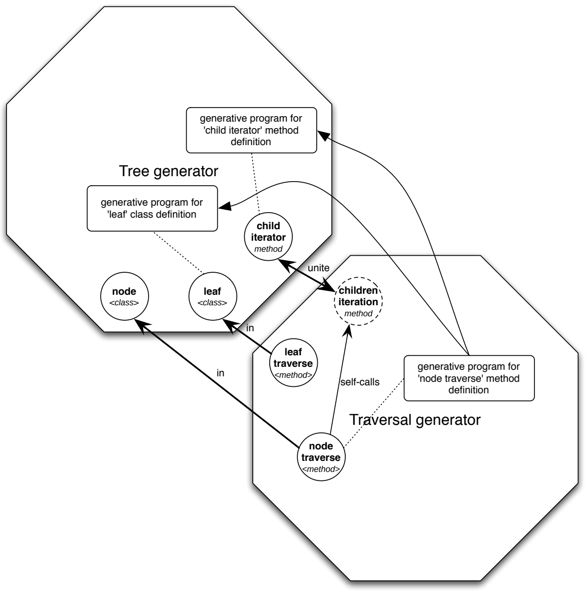

Tree generator
generative program for 'child iterator' method definition
generative program for 'leaf' class definition
child iterator method
node <class>
leaf <class>
unite
in
in
children iteration method
leaf traverse <method>
self-calls
generative program for 'node traverse' method definition
Traversal generator
node traverse <method>

Figure 5.11: Parameterization of the generative programs in the tree-traverse integrative composition.

a single argument (using the `singleKeyword/1` predicate). The traversal generator enforces this because the generated traversal implementation (i.e. the traversal methods) assume a single argument method. The generated traversal code that calls this method is shown below. The `do:` message represents the single argument message that is assumed by the traversal generator.

```
self do:[:child | child traverse]
```

The generative program that produces the node traversal `methods` listpart is shown below. It produces the traversal code for each node class that it needs to integrate methods with. Therefore it calls the `in/3` and `selfcalls/3` predicate to retrieve the node classes and the iteration method signature.

```
nodeTraverse(?listofMethods) if
   in(nodeTraverse,?,?listofNodes),
   selfcalls(nodeTraverse,childrenIteration,method(?,?iterator,?,?)),
   findall(method(?class,{traverse},{traverse ...
                                    self ?iterator [:child | child traverse]},
                              <?iterator>),
         member(class(?class,?),?listofNodes),
         ?listofMethods)
```

5.7.3 Adaptation for Integrative Composition

The traversal generator, as it was implemented in the previous section, is not very flexible. It requires that the method to retrieve the child nodes (generated by the tree generator) is a Smalltalk iterator method. However, the tree generator might very well be implemented to produce a method that returns the child nodes as a list. In such a case, the integrative composition of the generators fails. However, we can also implement the traversal generator such that it anticipates this integrative variability and adapts its generated program. The resulting traversal generator can then produce its generated program to work with a Smalltalk iterator method or with a method that returns a list of nodes. Therefore, the generative program for the `ChildrenIteration` required part is changed and becomes:

```
childrenIteration(method(?class,?selector,?body,?info) if
   unite(childrenIteration,?aPart,method(?class,?selector,?body,?info)).
```

The above generative program accepts that the program part integrates with any Smalltalk method. The selection of appropriate methods is now encoded in the generative program that produces the traversal code, which is shown below. The generative program now consists of an alternative for Smalltalk iterator methods and for methods that return the list of children, respectively. This consequently means an alternative for a single argument method or no arguments at all. This corresponds to a single keyword message and a unary message in Smalltalk, respectively. Any other kind of method results in a failure to generate the traversal code and thus in a failure of the integrative composition. The alternative is implemented in the logic program that implements the `makeIteration/2` predicate. The first alternative produces the iteration code for a single argument iterator method, the second alternative produces the iteration code for a method that returns a list of child nodes. The alternative is automatically chosen for a particular integrative composition.

```
nodeTraverse(?listofMethods) if
   in(nodeTraverse,?,?listofNodes),
   selfcalls(nodeTraverse,childrenIteration,method(?,?iterator,?,?)),
   makeIteration(?iterator,?iterationCode),
   findall(method(?class,{traverse},{traverse ?action ?iterationCode},<?selector>),
         and(member(class(?class,?),?listofNodes),traverseAction(?class,?action)),
         ?listofMethods)

makeIteration(?selector,{self ?selector [:child | child traverse]}) if
   singleKeyword(?selector).
makeIteration(?selector,{self ?selector do:[:child | child traverse]}) if
   unaryMessage(?selector).
```

5.8 Domain-specific Tree Integration

We now provide the implementation of the domain-specific tree integration example that was introduced in the previous chapter.

5.8.1 Tree Language Definition

The `Tree` language allows us to specify a tree structure. In such a specification, the possible names of nodes and leaves are declared, as well as some restrictions with respect to their composition. The definition of each node must specify its possible child nodes and leaves, as well as the fixed or maximum number of children that the node must or can have.

Representational Mapping of Program Parts

The possible program parts of a `Tree` language program are nodes and leaves, respectively represented as follows:

```
node(?name,?arity,?arityKind,?possibleChildren).
leaf(?name).
```

The representation of a node contains the name (`?name`), the number of children (in `?arity`) and the concrete names of the nodes and leaves that are allowed as children (in `?possibleChildren`). The number of children determines either a fixed number or a maximum number of children. This is determined by the `?arityKind`, which is either `fixed` or `maximum`, respectively. The representation of a leaf only contains the name (in `?name`).

Integration and Dependency Relations

The logic declarations that can be used to declare the `supportsChild` and `unite` integration and dependency relations are:

```
supportsChild(?node,?nodeOrLeaf)
unite(?nodeOrLeaf,?nodeOrLeaf)
```

The constraint that enforces the `supportsChild` dependency or integration relation is:

```
constraintSupportsChild(?nodeA,?nodeB,node(?nameA,?,?,?lA),node(?nameB,?,?,?))if
   member(?nameB,?lA).
constraintSupportsChild(?nodeA,?leafB,node(?nameA,?,?,?listA),leaf(?nameB)) if
   member(?nameB,?listA).
```

The first rule checks the `supportsChild` relation between a node and another node, while the second rule checks the constraint between a node and a leaf.

Composition Conflict Detection

The `Tree` language definition requires all nodes and leaves to have unique names. Consequently, in an integrated `Tree` language program, the only composition conflict that can exist is the definition of nodes or leaves with identical names. This is checked by the `uniquesConstraint`, which is shown below. The first three logic rules implement the case where there is no `unite` integration relation between the program parts. The last two logic rules implement the case where a `unite` integration exists. In the first case, no equal names may exist, while in the second case, equal names are required.

```
constraintDef(node,node,<uniquesConstraint>).
constraintDef(leaf,leaf,<uniquesConstraint>).
constraintDef(node,leaf,<uniquesConstraint>).

uniquesConstraint(?partnamA,?partnamB,node(?nodeA,?,?,?),node(?nodeB,?,?,?))  if
   not(unite(?partnamA,?partnamB)),
   not(equals(?  nodeA,?nodeB)).
uniquesConstraint(?partnameA,?partnameB,leaf(?leafA),leaf(?leafB)) if
   not(unite(?partnameA,?partnameB)),
   not(equals(?leafA,?leafB)).
uniquesConstraint(?,?,node(?nodeA,?,?,?),leaf(?leafB)) if
   not(equals(?nodeA,?leafB)).

uniquesConstraint(?partnameA,?partnameB,node(?name,?arity,?kind,?list),
                       node(?name,?arity,?kind,?list)) if
   unite(?partnameA,?partnameB).
uniquesConstraint(?partnameA,?partnameB,leaf(?name),leaf(?name)) if
   unite(?partnameA,?partnameB).
```

5.8.2 Generation of Tree Programs

Generators `C` and `D`, as shown in figure 4.14 in the previous chapter, contain a simple implementation that serves as an example. Generator `C` produces the program parts named `cLeaf` and `cNode`. Generator `D` produces program parts `dLeaf` and `dNode`. Their implementation is shown in table 5.7. The generative programs that produce the `cNode` and `dNode` program parts anticipate `supportsChild` integration relations to both leaves and nodes. The dependency relations that are defined in generators `C` and `D` are:

```
supportsChild(cNode,cLeaf).

supportsChild(dNode,dLeaf).
```

Part	Predicate	Declarations

cLeaf	`cLeaf/1`	`cLeaf(leaf(LeafC1)).`
		`cLeaf(leaf({LeafC1?leafName})) if`
		`  sunite(cLeaf,?,leaf(?leafName))`

cNode	`cNode/1`	`cNode(node(NodeC2,2,fixed,?children)) if`
		`  findall(?child,supportsChild(cNode,?,leaf(?child)),?leaves),`
		`  findall(?child,supportsChild(cNode,?,node(?child,?,?,?)),?nodes),`
		`  append(?leaves,?nodes,?children).`

dLeaf	`dLeaf/1`	`dLeaf(leaf(LeafD1)).`

dNode	`dNode/1`	`dNode(node(NodeD2,2,fixed,?children)) if`
		`  findall(?child,supportsChild(dNode,?,leaf(?child)),?leaves),`
		`  findall(?child,supportsChild(dNode,?,node(?child,?,?,?)),?nodes),`
		`  append(?leaves,?nodes,?children).`

Table 5.7: Generators `C` and `D` that produce a `Tree` program.

5.8.3 Integrative Composition

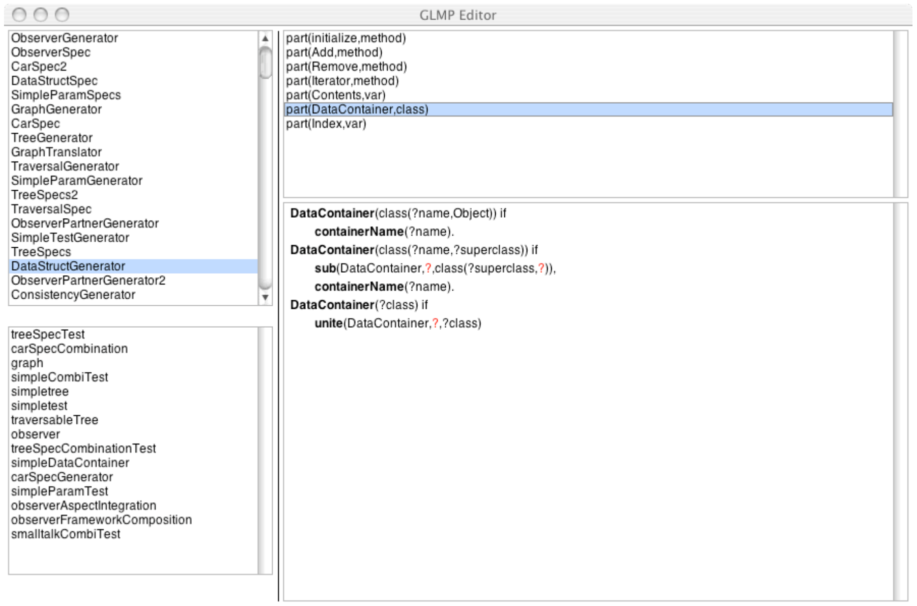
The integrative composition of generators `C` and `D`, shown in figure 4.14 in the previous chapter, is implemented using the following integration specification:

```
unite(cLeaf,dLeaf).
supportsChild(cNode,dNode).
```

The result of this integrative composition is an integrated program in the `Tree` language. This integrated `Tree` language program is then handed to the tree generator that produces an integrated tree implementation as depicted in figure 4.14 in the previous chapter. This integrated tree implementation is able to build integrated trees, which would not have been possible using a Smalltalk-level integrative composition.

5.9 Prototype GLMP System

To conduct our experiments, we have implemented a prototype implementation of the GLMP system. This prototype implements all functionality explained in this chapter and it allows the full implementation of program generators as we have shown them. Furthermore, it provides a simple user interface that facilitates the implementation of program generators. Figure 5.12 shows a screenshot of this user-interface. The windows shows the list of generators and for each generator, a list of program parts is shown. For each program part, the developer can write the generative program separately. By selecting multiple program parts in the list, the developer can read and write the dependency relations imposed on these parts. The integration specifications and language definitions are entirely implemented using editors provided with Soul.



```
part(initialize,method)
part(Add,method)
part(Remove,method)
part(Iterator,method)
part(Contents,var)
part(DataContainer,class)
part(Index,var)
```

```
DataContainer(class(?name,Object)) if
    containerName(?name).
DataContainer(class(?name,?superclass)) if
    sub(DataContainer,?,class(?superclass,?)),
    containerName(?name).
DataContainer(?class) if
    unite(DataContainer,?,?class)
```

Figure 5.12: User-interface of the GLMP Prototype Implementation.

5.10 Conclusion

We have described how integrative composable program generators are implemented in GLMP. The GLMP system accepts the implementation of program generators, input specifications, integrative compositions and language definitions in the form of logic metaprograms. We have described how a language definition and a program generator can be implemented as logic metaprograms and how LMP provides appropriate linguistic abstractions for the implementation of integrative variabilities.

All integrative variabilities in the implementation of an integrative composable generator are handled through alternative generated program parts. Logic programming provides us with the appropriate linguistic features of subsequent (alternative) results of a logic program. Furthermore, alternative results are easily implemented as multiple logic declarations that implement the same logic predicate. Each alternative logic declaration can implement the resolution to a composition conflict and the adaptations required for integration.

The GLMP system is a close interplay between the Soul logic evaluator and a constraint checker. To execute the generation and the integrative composition, the GLMP system builds a constraint network and executes the logic metaprograms to find a solution to this constraint network. The solution to the constraint network is the implementation of all program parts that adhere to the dependency and integration relations and that do not result in a defined composition conflict.

Chapter 6

Feature Description Language

The generative logic metaprogramming technique introduced in the previous chapters offers a general approach to design and implement integrative composable program generators. Until now, we have described integrative compositions in different (domain-specific) languages. In this chapter, we introduce a specific domain-specific language that can be used by many different generators: the Feature Description Language (FDL).

6.1 Introduction

The generative programming technique introduced in the previous chapters allows us to design and implement integrative composable generators. The composition conflict detection mechanism guarantees that no declared composition conflicts occur in the integration of the generated programs. We have shown some simple examples of integrations at the Smalltalk and Tree language levels. However, the declared composition conflicts are limited to conflicts that can be detected in the common modeling language(s). Many more interferences between the generated programs can exist. In order to detect all possible composition conflicts in an integrative composition, an appropriate hierarchy of levels of abstraction and modeling languages is necessary. In essence, we would need to make sure that each possible composition conflict can be declared in an appropriate modeling language and that all generators also use those languages in their implementation. This is, however, impossible to achieve in an open environment, where unanticipated compositions of program generators can occur and where generators are developed independently.

In contrast, a closed environment offers more opportunities. In the development of a library of program generators, a *master design plan* of modeling languages can be defined. With this master design plan, we mean that the developer has an overview of all possible generators and languages in the library. It means that an appropriate set of modeling languages can be designed such that all composition conflicts, that are defined in these languages, are detected. If all generators adhere to the use of the appropriate modeling languages to express their generated implementations, all composition conflicts can be declared, detected and even prevented. However, the design and implementation of appropriate modeling languages that express each possible concern is still a complex and tedious task. The definition of all composition conflicts will most likely be an even more tedious task. The effort to define appropriate languages can be reduced by the use of a so-called *feature description language* [vDK02]. Instead of implementing an entire domain-specific modeling language, many program generators accept a *feature description* as an input specification. A well know example of such generators are the GenVoca generators that produce a program according to the set of features that is defined by a GenVoca expression. Although the GenVoca language itself is common to all GenVoca generators, the possible feature descriptions that can be requested are different from one generator to another. Similarly, the feature description language (FDL), which is explained in this chapter, is defined once for all generators and each generator defines the valid feature descriptions that it can generate. This approach allows us to build a library of program generators that share a common modeling language (the FDL) and where high-level composition conflicts can be detected without the need for individual modeling languages.

The following section provides a brief introduction to feature diagrams. These feature diagrams are used to describe the features that a program generator can produce. The description of these feature diagrams needs to be expressed in the Feature Description Language (FDL), which is explained in section 6.3. We present the implementation of the FDL in GLMP in section 6.4, followed by how an integrative composition of program generators occurs when an integration is specified at the FDL level in section 6.5.

6.2 Feature Diagrams

The development of any generator is preceded by a domain analysis. This domain analysis results in a feature model that describes the commonalities, variabilities and dependencies of the separate features that can be included in the generated program. A well known domain analysis methodology is FODA (Feature-oriented Domain Analysis) [ea90]. An important part of the feature model in FODA is the feature diagram, which is a graphical notation for describing the (variable) features and their dependencies. An example of such a feature diagram is given in figure 6.1. This diagram is a well known FODA diagram example of a car (adapted from [Cza98, vDK02]). We describe the notation of feature diagrams by means of this example. The diagram describes that a car contains the features `Carbody`, `Transmission`, `Engine`, `Horsepower` and `Trailerhook`. The first four features are mandatory features that should be included in each car. The `TrailerHook` feature is optional. The `Transmission`, `Engine` and `Horsepower` features are *composite* features while the `TrailerHook`, `Carbody`, `Gasoline` and others are *atomic* features. Composite features consist of other features, while atomic features do not. Furthermore, the `Transmission` feature, for example, consists of the mutually exclusive features `Manual` and `Automatic`. This means that a transmission can be automatic or manual, but not both. The `Engine` feature is also a composite feature but consists of the non-mutually exclusive features `Electric` and `Gasoline`. It means that the car's engine can be electric, gasoline powered or a combination of both.

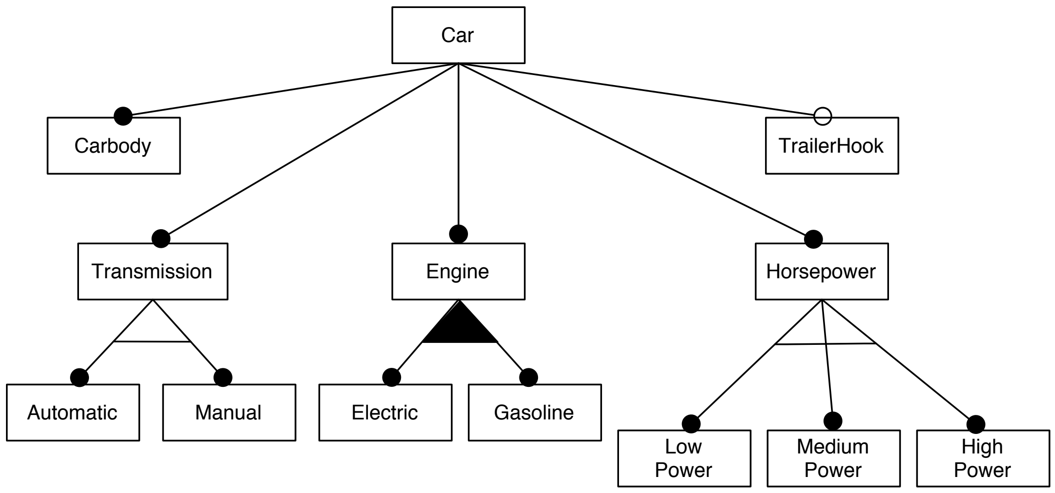


Figure 6.1: Feature diagram of a car.

A feature diagram describes the set of valid *featural descriptions* of a software system [Cza98]. A featural description is a set of atomic features that describes a particular configuration. In other words, a feature diagram describes all possible valid configurations of a concept, with respect to its possible features. In the example above, the feature diagram describes all possible configurations of a car. A valid featural description (or configuration) is `CarBody,Automatic,Electric,Medium Power`. Program generators accept such a featural description and produce a generated program that implements the requested features. The feature diagram is thus defined by the developer of the program generator. Each program generator must also check if the provided featural description is valid. In our approach, we have provided a common language to express the feature diagrams of each generator, which eases the development of generators. In essence, the language implementation is common to all generators. Each generator merely needs to describe the feature diagram to 'configure' the language. For this purpose, a language to describe feature diagrams is required. This is described in the following subsection.

6.3 Feature Description Language

Van Deursen and Klint [vDK02] have developed a textual notation for feature diagrams, which they call the *Feature Description Language* (FDL). We will adopt their language in a logic notation to express feature diagrams in the implementation of program generators. Each program generator describes all possible featural descriptions using a feature diagram expressed in our logic variant of FDL. This means that FDL is used as a common modeling language for a range of generators. In this section, we describe how program generators can express a feature diagram in FDL.

Each feature that is described in a feature diagram is represented by a logic declaration. The predicate of that logic declaration determines the relation (mutually exclusive, mandatory,...) between these features:

Mandatory features These features need to be included in each featural description. They are defined using the `all` predicate:
`all(?featureName,?listOfMandatoryFeatures)`

Mutually exclusive features Only one of these features may be included in a featural description. They are defined using the `oneOf` predicate:
`oneOf(?featureName,?listOfMutuallyExclusiveFeatures)`

Non-exclusive features A subset of these features should be included in a featural description. They are defined using the `moreOf` predicate:
`moreOf(?featureName,?listOfMutuallyExclusiveFeatures)`

Optional features The feature is optional. It should not be included, but can. This is defined using the `optional` predicate:
`optional(?featureName)`

Figure 6.2 shows the implementation of the car feature diagram of figure 6.1 in our logic variant of the FDL language. Each generator that accepts a featural description should provide such a specification of its feature diagram. Given this description, the featural descriptions can be checked for correctness. We will explain later on that this description steers the conflict detection constraints. First, we describe how featural descriptions are represented in generative logic metaprogramming (GLMP)

```
all(Car,<Carbody,Transmission,Engine,Horsepower,TrailerHook>).
optional(Trailerhook)
moreOf(Engine,<Electric,Gasoline>).
oneOf(transmission,<Automatic,Manual>).
oneOf(horsepower,<LowPower,MediumPower,HighPower>).
```

Figure 6.2: Feature diagram described in the logic version of FDL.

in the following section. For clarity, we will use the car example to illustrate the entire FDL language.

6.4 GLMP Feature Description Language

In the GLMP system, the FDL language is defined similarly to all other domain-specific languages. Its implementation is somewhat more complex because it is itself parameterized with a feature diagram description, as shown in the previous section. Furthermore, the integrative compositions in terms of the FDL are also different from domain-specific integrative compositions shown in the previous chapters.

6.4.1 Program Parts

The programs in FDL are featural descriptions. These featural descriptions are the input programs for generators that produce a Smalltalk implementation. Once again, concrete syntax of featural descriptions is not important. Because they are essentialy nothing more than a list of features [Cza98], they can be described by means of simple lists, such as in [vDK02]. Therefore, a featural description in GLMP is a set of feature names.

There is only a single type of program part in FDL: `features`. It must contain an entire featural description. The representation of a `features` program part in GLMP is:

```
features(?name,?featureNameList)
```

The `?name` is a name to identify the featural description. This is required because a single program generator can accept multiple featural descriptions. The list of features (`?featureNameList`) must contain only atomic features. For example:

```
features(myCar,<CarBody,Automatic,Electric,MediumPower>)
```

is a featural description for a car. The program part

```
    features(hisCar,<CarBody,Automatic,Gasoline,MediumPower>)
```

is a program part that contains the featural description of another car (the `?name` is different).

Featural descriptions can be generated by program generators as well. This is the case in a translational composition of programs generators, i.e. a program generator produces a featural description that is the input specification of another generator. Furthermore, these featural descriptions can also be integrated. Consequently, a generator will also produce alternative featural descriptions. We will explain later on how this can drive the composition conflict detection and resolution mechanism.

6.4.2 Integration and Dependency Relations

In FDL, the integration and dependency relations are identical. The program parts of FDL can be related through four relations: `requires(x,y)`, `excludes(x,y)`, `compatible(x)` and `unite`. These relations are parameterized and they can be drawn between program parts to enforce the inclusion or exclusion of particular features, depending on the features declared in one of the two parts. They are illustrated in figure 6.3 and described below:

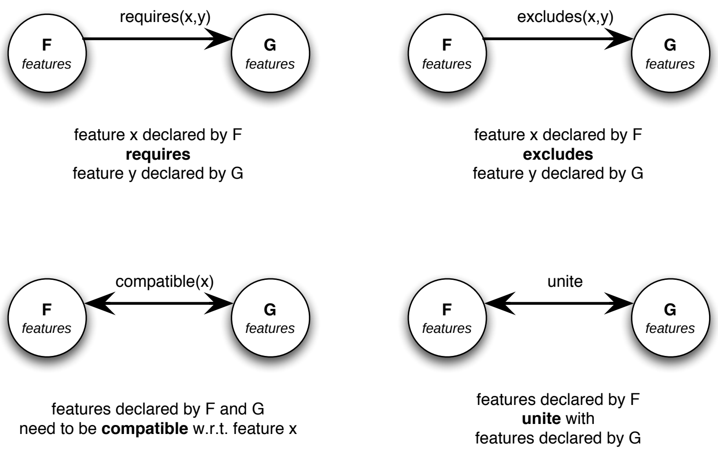

requires(x,y)
F features
G features
feature x declared by F requires feature y declared by G
excludes(x,y)
F features
G features
feature x declared by F excludes feature y declared by G
compatible(x)
F features
G features
features declared by F and G need to be compatible w.r.t. feature x
unite
F features
G features
features declared by F unite with features declared by G

Figure 6.3: Dependency Relations in FDL.

Requires A `requires(x,y)` relation enforces that if feature `x` is declared in the origin part of the relation, then feature `y` must be declared in the destination part of the relation. Figure 6.4 is an example of a generator that produces a featural description of a car. It shows the use of a `requires` dependency relation between separate featural descriptions. We have also included some simple generative programs that produce the featural descriptions.

Excludes The `excludes(x,y)` relation enforces that if feature `x` is declared in the origin part, then feature `y` may not be declared in the destination program part.

Compatible The `compatible(x)` integration relation enforces a compatibility of the featural descriptions with respect to feature `x`. Feature `x` can be a composite as well as an atomic feature. Featural descriptions are compatible with respect to a particular feature `x` if the subfeatures of that feature can exist in the same featural description, without violating the constraints imposed by the feature diagram. For example, consider the use of the `compatible` relation in figure 6.5. The `compatible(transmission)` relation is declared between the parts named `Executive Car` and `Employee Car`. These parts contain the featural descriptions of different cars, produced by different program generators. Because the first generator produces the featural descriptions for executive cars and the second generator produces the featural descriptions for employee cars, this integration relation enforces that employee and executive cars have the same kind of transmission. Consequently, in this artificial example, both 'generators' that accept this featural description as an input specification will produce a car such that both cars either have a manual transmission or an automatic transmission.

Unite The `unite` relation has an identical meaning as in the Smalltalk language. It enforces that the program parts are *identical* and thus consequently *unite* in the same generated program part.

These relations can be explicitly included in the implementation of a program generator to enforce dependencies between separate featural descriptions. Besides the explicit relations, the same kind of dependencies are automatically enforced in a single featural description. These 'implicit' dependencies are automatically derived from the feature diagrams described in FDL. For example, the car feature diagram enforces that if the feature `Automatic` is chosen for the transmission, this excludes the feature `Manual`. Consequently, a featural description that declares both features is not a correct program part. The explicit dependency relations allow us to specify custom dependencies between separate featural descriptions. For example, we can enforce the `requires(HighPower,HighPower)` dependency between featural descriptions of different cars. We could even enforce dependencies between featural descriptions that are an instantiation of different feature diagrams.

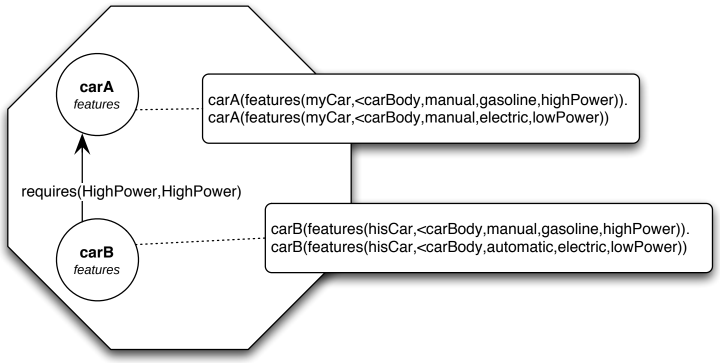

carA
features
carA(features(myCar,<carBody,manual,gasoline,highPower)).
carA(features(myCar,<carBody,manual,electric,lowPower))
requires(HighPower,HighPower)
carB
features
carB(features(hisCar,<carBody,manual,gasoline,highPower)).
carB(features(hisCar,<carBody,automatic,electric,lowPower))

Figure 6.4: A generator that produces a car featural description.

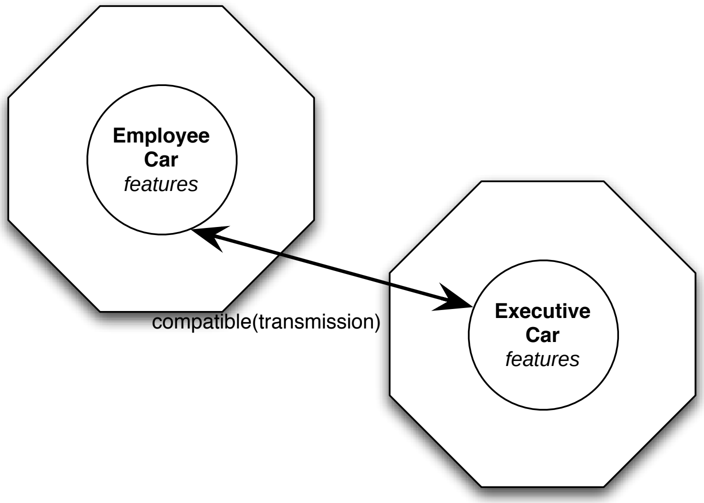


Figure 6.5: An integration of featural descriptions for cars.

6.4.3 Composition Conflict Detection

There are no separate composition conflict detection constraints defined for FDL. Instead, the composition conflicts are detected by violations of the constraints imposed by the feature diagram and violations of the dependency and integration relations. The complete implementation of the FDL in GLMP can be found in appendix A. We will now describe how this feature description language provides the ability to detect high-level composition conflicts.

6.5 Integrative Compositions

An integration of featural descriptions does not integrate the actual generated implementations. It merely serves as an enforcement mechanism for the compatibility of features in an actual integrative composition at the Smalltalk level. Therefore, the integrative compositions are somewhat different than those shown in the previous chapters.

The integrative composition is not only expressed at the FDL level but also at the Smalltalk level. This means that the composition interfaces of the generators in our library can consist of both FDL and Smalltalk program parts. The presence of FDL and Smalltalk program parts in a single composition interface is specifically true for composed generators or generators that are specifically built for an integrative composition with other generators. Figure 6.6 shows how a composed generator's composition interface can consist of both FDL program parts and Smalltalk program parts. The illustration shows an integrative composition of two composed generators (`A` and `B`). Each of the composed generators `A` and `B` is built as a translation composition of other generators (e.g. a translation composition of generators `A1` and `A2`). The integrative composition of `A` and `B` is specified in terms of the featural descriptions in FDL (i.e. FDL `A1` and FDL `B1`) **and** in terms of the Smalltalk program parts.The composed generator that results from the integrative composition of generator `A` and `B` produces a Smalltalk program that is an integration of the Smalltalk programs produced by generators `A2` and `B2` and enforces that the featural descriptions produced by generators `A1` and `B1` adhere to particular dependencies (as specified in the integrative composition).

The composed generator `A+B` accepts a featural description in FDL `A1` and FDL `B1`. These FDLs are different because they are configured for different feature diagrams (i.e. the different feature diagrams of the different generators). Of course, the feature description languages can also be the same language. They can even be the same as FDLs `A2` and `B2`. The scheme in figure 6.6 shows the most general integrative composition possible. In essence, the input featural descriptions in FDL `A1` and `B1` are translated to featural descriptions in FDL `A2` and `B2`, which are subjected

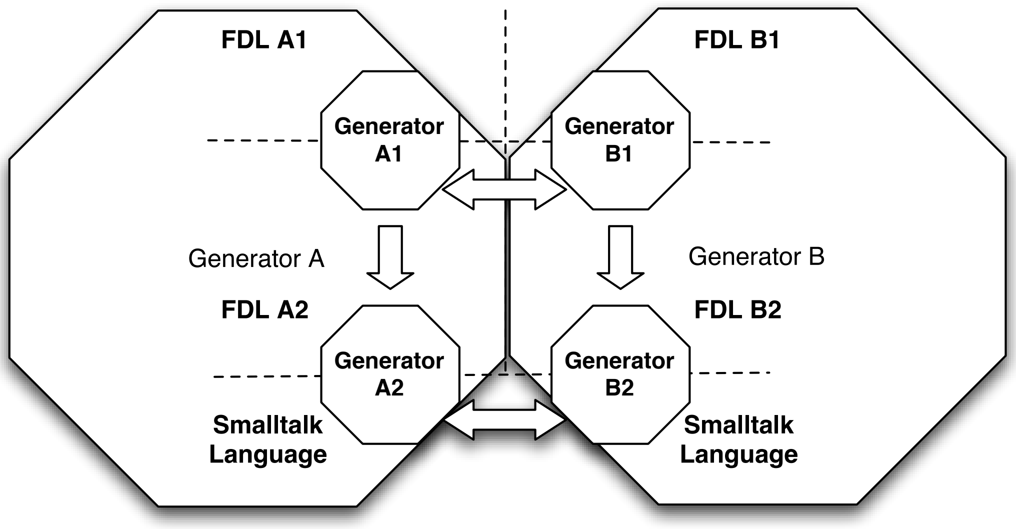


Figure 6.6: Integrative Composition scheme for generators using FDL.

to an integrative composition. This means that the integration enforces particular dependencies. If these dependencies are not met, generators `A1` and `B1` should provide alternatives that do meet the integration relations. If a correct set of featural descriptions in FDL `A1` and `B1` is found, they are handed to generators `A2` and `B2` that generate the Smalltalk programs, which are integrated as specified in the integration relations. Evidently, this requires that generators that produce FDL programs also anticipate the possible composition conflicts. The possible composition conflicts are known because the feature diagrams are known. Resolution of the composition conflicts requires to adapt the featural description by providing an alternative one. Of course, this results in a different functionality of the final generated program, which is not what we want. Nevertheless, we can use the mechanism of alternatives to resolve composition conflicts because a generator often does not produce a single featural description. This is illustrated in figure 6.7. In this composition, generators `A2,A3` and `B2,B3`, respectively, are provided with featural descriptions as input. An integrative composition can result in a composition conflict for one of these featural descriptions. If possible, the generators `A1` and `A2` anticipate the composition conflict and provide an alternative featural description that circumvents it. This means that *a part* of the final generated programs has a different functionality. Therefore, the internal dependency relations of `A1` and `A2` must enforce a change to the other featural descriptions that are produced such that the *entire* final generated program still has the same functionality. An example of such an integrative composition is shown in the following chapter.

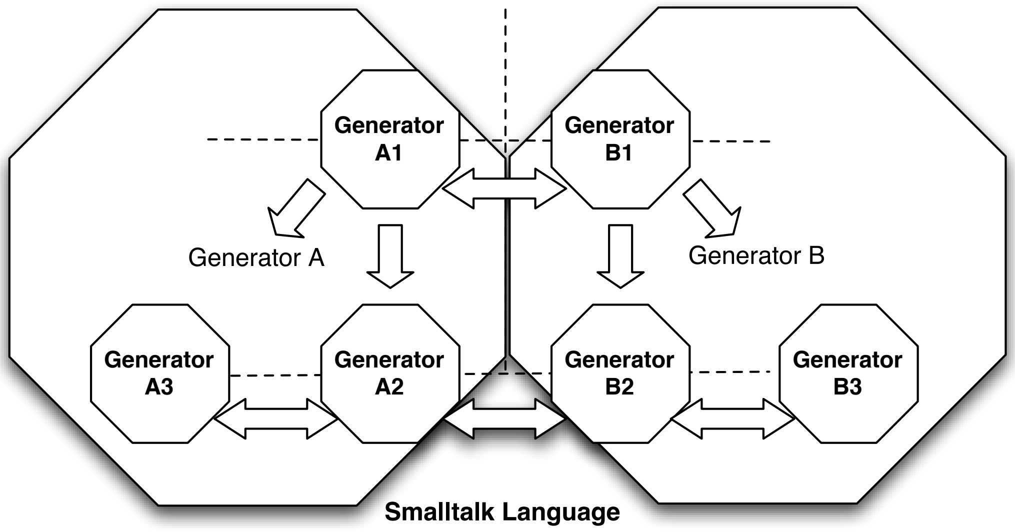


Figure 6.7: Integrative composition influences generated code of generators `A3` and `B3`.

6.6 Conclusion

We have described the Feature Description Language (FDL) as an alternative to the definition of a set of domain-specific languages. The FDL does not allow domain-specific integrative compositions but does allow us to detect domain-specific (or high-level) composition conflicts between generators of a single library. An integrative composition using FDL is expressed both at the FDL and the Smalltalk level. Composition conflicts in FDL are detected as violations of the constraints imposed by the feature diagrams or the integration and dependency relations. Composition conflicts can be resolved because a program generator anticipates to the possible composition conflicts in FDL and adapts the featural description and generated Smalltalk programs accordingly to circumvent the composition conflict.

We will use the FDL extensively in the implementation of a library of program generators, presented in the following chapter.

Chapter 7

A Library of Generators

In this chapter, we validate our approach by implementing a small library of integrative composable program generators. We show the flexibility of integrative compositions at the Smalltalk level, implement a generator that exploits the featural descriptions to resolve composition conflicts and we implement generators that compose through domain-specific integrative composition.

7.1 Introduction: Library Context

A company produces a range of tools that are adapted to a client's requirements. One kind of tools is aimed at document processing, according to particular requirements. Such *Integrated Document Processing Tools* are customized text editors and document processors focused at the treatment of a particular kind of documents, specific to a business process. The client provides a set of requirements for documents and the company produces a tool that adheres to these requirements. This means that the tool allows the editing and creation of documents in a particular style, lay-out, etc... in disciplined ways (e.g. only by editing forms). Documents are also stored on appropriate databases or fileservers, for collaborative editing.

The exact usage and capabilities of these tools are not important in this dissertation. They merely provide a scope for the development of a library of program generators. The tools are produced by means of generators in this library. What is important is that clients of this company order tools customized to their technical and functional requirements. Functional requirements are, e.g. the kind of documents or data and how they should be processed in the document processing tool, the lay-out of the user interface, the style and lay-out of the document, etc.... Technical requirements are, for example: how the documents should be stored (database, textfile,...). Instead of building a new tool from scratch each time a client orders one, it is quite obvious that the developers will reuse generic implementations that they adapt to the requirements of the client. However, many different possible feature variations exist in many of the reusable components (e.g. parsers, data containers, user interface, etc. . .). Consequently, they cannot be implemented in an efficient and scalable way by means of a library of fixed components or by means of frameworks, without still requiring a lot of repetitive and manual implementation work. Instead of developing a reusable component for each possible set of requirements, the developers decide to prevent running into the library scaling problem [Big94] by adopting a program generation approach.

A generator is implemented for some of the reusable components they require. We briefly introduce these generators here. Their variabilities, implementations and compositions are discussed in following sections.

Data Container generator produces a data container implementation to store objects, based on a description of requirements.

User Interface generator produces an implementation for a user interface, based on a description of this user interface.

Observer generator produces an implementation for an observer-observable collaboration, based on a description of requirements.

Data Consistency generator produces an implementation of a collaboration between different generated data containers to keep their contents consistent.

In addition to these elementary program generators, the company also builds composed generators for its integrated document processing tools. This is because such a tool is also a software product that benefits from an implementation as a generator. Instead of building a monolithic generator for an entire integrated document processing tool, they build it as a composition of generators in the library. The same generators are thus reused in the implementation of the different tool generators, which are:

Document Editor generator produces an editor, customized to the user's requirements, that allows to edit an entire document.

Forms Editor generator produces an editor that allows creating and editing documents by filling in a set of forms. The tool creates the document using information entered in these forms.

Integrated Editor generator produces a combined tool that allows to edit a document using the forms editor and the document editor simultaneously.

Because the company produces its own library of generators, it would be possible to identify and use an appropriate set of modeling languages in the implementation of each generator such that they can be composed at an appropriate domain-specific level. As a consequence, all composition conflicts could also be declared in the appropriate language. However, designing and implementing an appropriate set of modeling languages to capture each concern and all possible composition conflicts is quite a complex and tedious task. However, it is required to declare and detect composition conflicts between generators. Therefore, in the development of this library we extensively use the FDL language.

Overview

The following section describes the implementation and some example integrative compositions of the `Data Container` generator. Section 7.3 introduces the implementation of the `Observer` generator and elaborates on a set of possible integrative compositions with this generator. The `User Interface` generator is described in section 7.4. At that time, all atomic generators are introduced and we then describe some composed generators in section 7.5. We describe some integrative compositions of the previous generators and also describe the implementation of the `Consistency` generator, which is specifically targeted at an integrative composition with the `Data Container` generators. We also describe the building of the `Document Editor`, `Forms Editor` and `Integrated Editor` generators through integrative compositions at the Smalltalk, FDL and Tree language levels. In a final section (7.6), we evaluate the experiments and discuss the impact of the generative technique on other characteristics of program generators.

7.2 Data Container Generator

The `Data Container` generator produces a customized implementation for a data container. Because we want to focus on the integrative compositions and not the implementation of functional variabilities, the data container implementation that is generated is rather simple. It allows to collect a number of values (objects), allows to remove them and provides a method to iterate over the contained values. The feature diagram in figure 7.1 shows the possible feature variations between the data containers that this generator can produce. The `Storage` feature describes the choice wether the data container keeps a reference to the originally added object (`Reference`), or wether it only stores a copy of it (`Copy`). The `Size` feature allows choosing that a data container instance grows automatically in size (`Growable`) or that it has a fixed size (`Fixed`). For the `Data Management` feature, it can be requested that the data container does not store duplicates of objects using the `Uniqueness` feature.

If the `Duplicates` feature is included in a featural description, the data container will allow to store the same object multiple times. A last feature (`Indexing`) is concerned with how the objects are stored and removed from the data container. Choosing the `KeyValue` feature means that the data container stores key-value pairs. As a consequence, the addition method requires a key and a value as arguments and the removal method accepts a key to identify the object to be removed. In essence, the data container then implements a dictionary. If the `Plain` feature is chosen, the container keeps a plain collection of the objects. The addition and removal methods then have a single argument, i.e. the object to be removed or stored. The features `Storage`, `Size`,`Data Management` and `Indexing` all have mutually exclusive subfeatures. An additional dependency is included between the `Copy` and `KeyValue` features. If the `Copy` feature is included in a featural description, then the `KeyValue` feature should be chosen as well. This is because if a plain container copies the objects upon addition, all identification of that object is lost. This means that we cannot remove the object from the container, given the original object that is provided as an argument to the removal method. Therefore, in case of copying the original objects, a key should be associated with the object, such that we can remove the object, given the key. Another additional dependency sets that if the `Uniqueness` feature is chosen, then the `Reference` feature must be chosen as well. The implementation of this diagram in FDL is shown in figure 7.2. This is how a program generator includes the feature diagram description in the common featural description language.

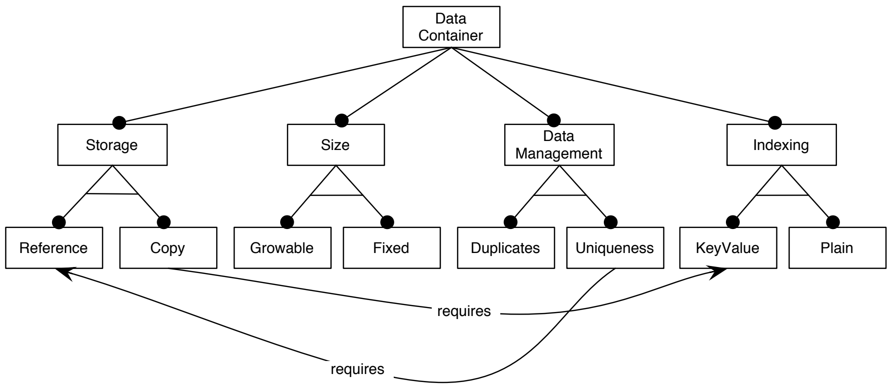


Figure 7.1: Feature Diagram of a Data Container.

The generated implementation for a data container consists of a single class with the appropriate methods implemented on it. An illustration of the generator's program parts, their dependencies, the composition interface and a sample generated program

```
all(DataContainer,<Storage,Size,DataManagement,Indexing>).
oneOf(Storage,<Reference,Copy>).
oneOf(Size,<Growable,Fixed>).
oneOf(DataManagement,<Duplicates,Uniqueness>).
oneOf(Indexing,<KeyValue,Plain>).
requires(Copy,KeyValue).
requires(Uniqueness,Reference).
```

Figure 7.2: Feature diagram of the Data Container Generator described in the logic version of FDL.

is provided in figure 7.3. The `DataContainer` program part is the class definition. There are two private program parts, i.e. the internal variables. The initialization (constructor), addition, removal and iterator methods are public program parts. All internal dependency relations in this generator express that the methods and variables are included `in` the class and that the methods `refer` to the variables.

7.2.1 Integrative Compositions

The data container generator produces a simple program. Moreover, a data container is almost always used as a standalone component in an application. In other words, an application developer will address the data container via the public interface, as any normal library component. This holds for any generated program that wants to use the data container. Nevertheless, even for such a simple generator, we identified some useful integrative compositions with other generators.

Subclass Another program generator (call it generator `X`) might produce a specialization for the data container class. This specialization class needs to subclass the generated data container class. However, the developer of generator `X` does not know all implementation details of the data container class. Consequently, if generator X merely generates its class as a subclass of a generated data container class, some methods might be inadvertently overridden or variables can be shadowed. Furthermore, the exact name of the data container class can vary from one generated implementation to another (because of name conflicts). For this purpose, it is useful that generator X is integratively composed with the data container generator. A `subclass(ClassX,DataContainer)` integration relation ensures that the generated class in the part named `ClassX` of generator `X` is effectively a subclass of the generated data container class. The composition conflict detection mechanism for the Smalltalk language ensures that no methods are overridden or variables shadowed unless it is specified with an `override`

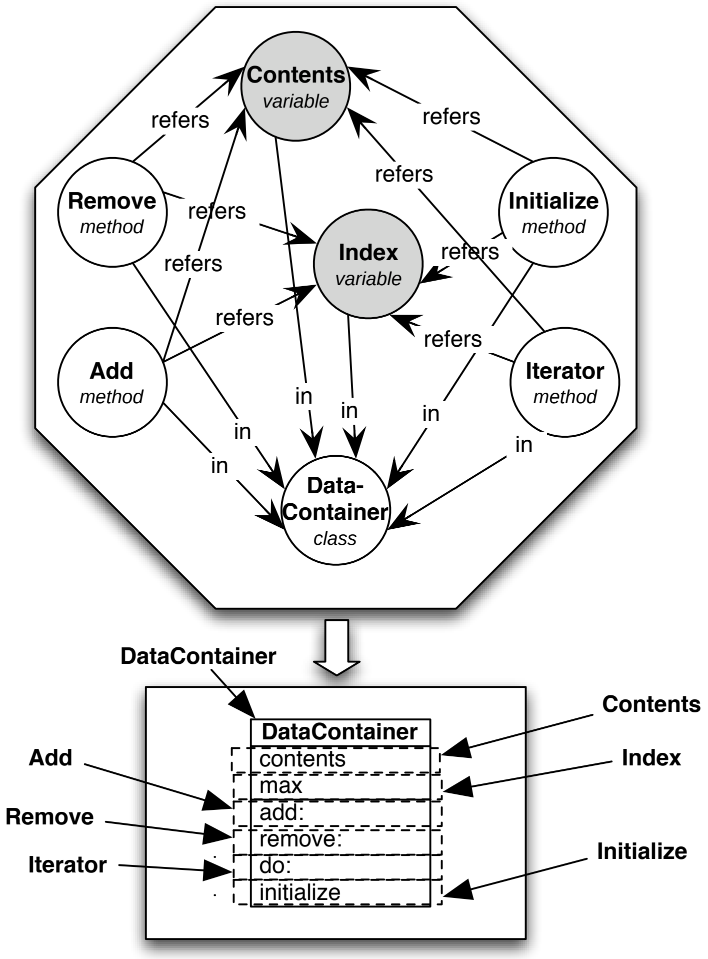


Figure 7.3: Data Container Generator with a possible generated program.

or `unite` integration relationship.

Method Integration with Required Parts Generator `X` can also be implemented with the intention of an integrative composition with the data container generator. For such a purpose, generator `X` includes a number of required parts. These parts represent the implementation class of the data container and the methods that the generated implementation produced by `X` needs to call on the data container class. This is useful because the name of the generated data container class and the names of the methods might differ from one generated implementation to another because a naming conflict has been resolved. The integrative composition thus requires to `unite` these required parts with their concrete counterparts declared in the composition interface of the data container generator. The integration relations for generator `X` and the data container generator are:

```
unite(ContainerClassX,DataContainer).
unite(ContainerAddX,Add).
unite(ContainerRemoveX,Remove)
```

Unite If the generated class of generator `X` already is a subclass of another class, it is impossible to integrate that class with the data container class through a `subclass` integration relation. In such a case, it might suffice to `unite` the generated class with the data container class. The result is that all necessary methods and variables are automatically included in the same generated class. If a `unite(ContainerClassX,DataContainer)` integration relation is declared, the integration relation propagation mechanism ensures that all methods and variables of both classes are integrated in the same class. The conflict detection mechanism also ensures that no methods or variables have the same name.

7.2.2 Internal Implementation

The data container generator anticipates integrative compositions and provides alternative implementations for each generated program part. To illustrate these anticipations for the data container generator, we include the generative program for the `Add` generated method part in figure 7.4. For simplicity, we omit the implementation of the functional variability according to the requested features. An example of such an implementation will be given in the description of the observer generator in the next section. The generative program shows how the generated method part is parameterized by the `refers` dependency relations to the `Contents` and `Index` parts and by the in relation to any part. This last relation does not specify its destination part's name. Because of this, it automatically matches a possible in integration relation as well as the dependency relation shown in figure 7.3. Mind that this means that the in predicate can have multiple alternative results, one for the presence of each in relation. In this way, the generative program anticipates the presence of an in integration relation. The generative system enforces the selection of the correct alternative result by verifying if the generated part adheres to the integration and dependency relations. Also mind that because of integration relation propagation, the DataContainer part is automatically involved in a unite integration relation with the appropriate part. Last but not least, the generative program allows to integrate a method body after its own body through the includeAfter integration relation.

```
add(method(?class,{add:},{add:anObject ...?afterBody},?completeInfo)) if
   refers(add,contents,var(?contents,?class)),
   refers(add,index,<var(?index,?class)>),
   in(add,?,class(?class,?)),
   includeAfter(add,{add:},<anObject>,?afterBody,?info),
   append(?info,<?contents,?index>,?completeInfo)
```

Figure 7.4: The Generative Logic Metaprogram for the Add method part.

The complete implementation of the data container generator can be found in appendix C. In this implementation, it can be observed that the generated implementation only uses the Smalltalk Array and Association classes.

7.3 Observer Generator

The observer generator produces an implementation for an observer-observable collaboration. The intention of such a collaboration is the implementation of a one-to-many dependency relation between objects so that when one object changes state, all its observers (dependents) are notified and updated automatically. It is also known as 'publish-subscribe', 'dependents' or as the 'observer pattern' [GHJV95]. An observer-observable collaboration is a typical program part that is frequently used and re-used in many different applications. This is illustrated by its adoption as a design pattern in [GHJV95]. However, most of the time, it is not implemented through generation but as an object-oriented framework. Of course, even though it remains a simple example, generation of an observer-observable implementation has some advantages. This is because even a simple collaboration such as observer-observable can have some required variabilities in its implementation. The possible variabilities that can be generated by the observer generator are illustrated in the feature diagram in figure 7.5. A first variability is concerned with the notification of observers by an observable in the `Notification` feature. The `Multi` feature means that a single observer object that subscribes multiple times will be notified an equal amount of times in case of a state change. The `Single` feature means that a single observer that subscribes multiple times will only be notified once. The `Single` and `Multi` features are mutually exclusive. A second variability is concerned with how an observer knows about the observable that it receives notifications from. The `Parameter` feature means that the observable object will send itself as an argument in the notification message to the observer. The `Instance Variable` feature means that the observer can keep a reference to its observable in an instance variable. Both these features can be present in a single implementation. The `Parameter` feature is often required if a single observer object can be notified by multiple observables. Because the observable sends itself along with the notification message, the observer can identify the originator of the notification message. The `Instance Variable` feature is often more practical in the implementation of an observer.

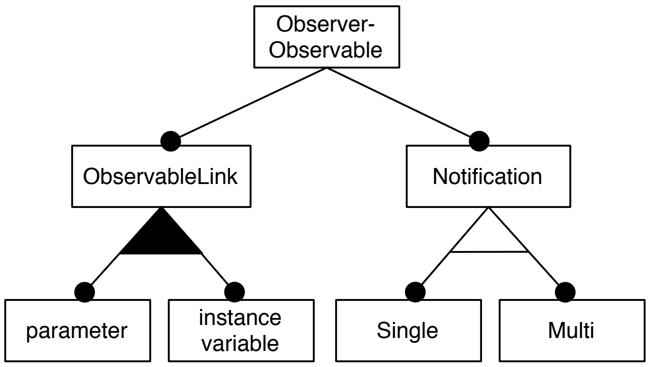


Figure 7.5: Feature Diagram of an Observer.

An example of a generated implementation together with an illustration of the observer generator is depicted in figure 7.6. The generator's composition interface and its internal dependency relations are shown. In this generator, all generated program parts are exposed through the composition interface. The generator actually produces a framework implementation of the observer-observable collaboration, in accordance with the requested features. It thus produces two classes, one for an `Observer` and another class for the `Observable`. In this particular generated implementation, the

`Observer` class contains an instance variable that contains a reference to an observable[1]. Furthermore, an initial implementation of the `update:` method is provided, which does nothing. In the `Observable` class, an instance variable is created to contain references to all observers and the necessary `addObserver:` and `delObserver:` methods are generated to add and remove observers from the observable, respectively. The `changed:` method implements the notification process.

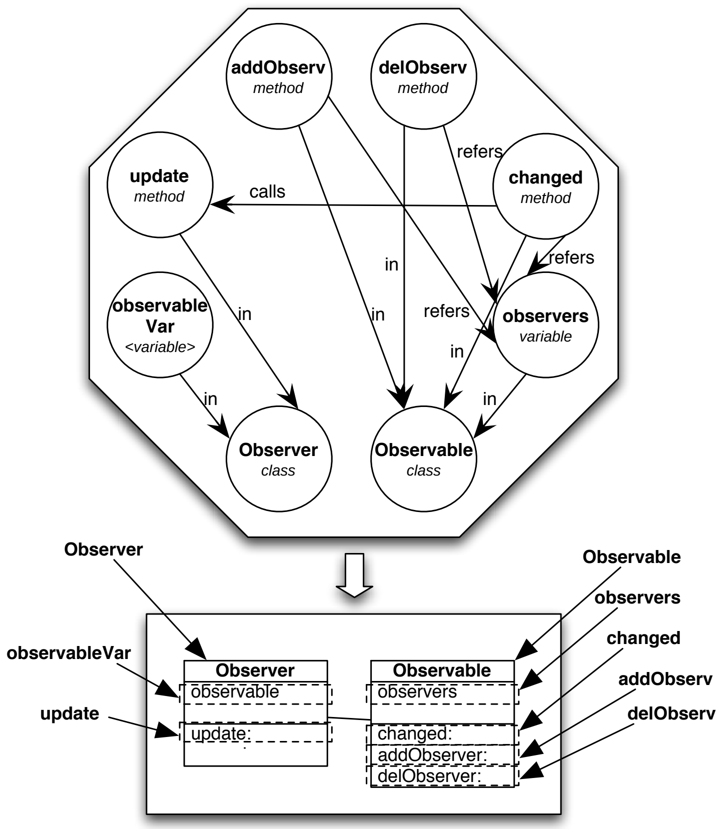


Figure 7.6: Observer Generator with a possible generated program.

[1]This means that the `Instance Variable` feature is implemented. If it was not desired, there would be no instance variable here.

7.3.1 Integrative Compositions

The generated observer implementation can integrate with other generated programs in multiple ways. This is, of course, achieved by an integrative composition of the corresponding generators. We describe the most common integrative compositions below. In all integrative compositions, the observer generator is composed with a generator that produces a concrete observable and concrete observers.

Framework Specialization Perhaps the most common integrative composition that one would want is one that boils down to a specialization of the framework that is generated by the observer generator. Figure 7.7 visualizes such an integrative composition [2]. The observer generator produces the framework and the other generator integrates its generated concrete observers and observables by subclassing. The figure also visualizes the generated parts of the other generator and their internal dependencies. The advantage of this kind of composition is that multiple concrete observer classes can be generated by the other generator. It also allows the inclusion of more generators in the integrative composition that integrate other observers and observables with the same generated framework (e.g. the `ObserverC` class in the figure).

Aspectual Integration Another integrative composition is illustrated in figure 7.8. This integrative composition merges a single concrete observer and a single concrete observable class with the observer-observable collaboration. The concrete observer is effectively integrated *into* the observer class produced by the observer generator. In essence, it means that the actual `Observer` and `Observable` classes are extended with additional methods and consequently, have their interfaces changed. A particularity in this integrative composition is the generated `myUpdate` method part. In the framework specialization, this method overrides the method in the `update` method part. In this aspectual integration, these methods `unite`. To make this integration relation work, the generation of the `update` method part simply copies the implementation of the `myUpdate` method part. This satisfies the `unite` integration relation and achieves the integrative composition that we want. Indeed, the `myUpdate` method part contains the functional implementation, while the `update` method part that is normally produced by the observer generator contains an empty implementation. Although this aspectual integration means that the observer-observable implementation can only be integrated with a single concrete observer and observable, such an aspectual integration can be useful. It provides the opportunity to insert the observer collaboration in classes that cannot subclass from the generated implementation.

[2]In the figure, the lay-out of the observer generator's generated parts has been changed a little to fit the lay-out of the integration relations.

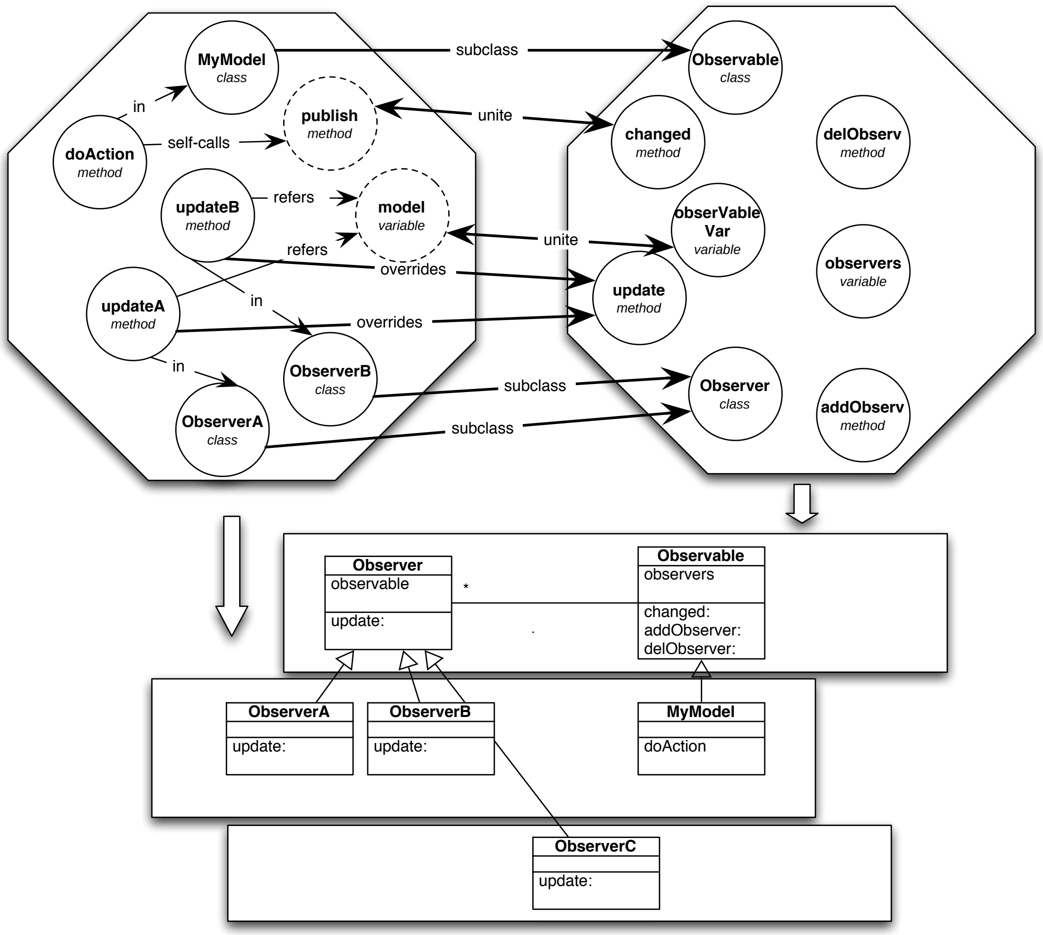


Figure 7.7: Observer generator in an integrative composition that results in framework specialization.

This is often the case when, for example, the concrete observers also integrate with other classes than those in the observer-observable implementation.

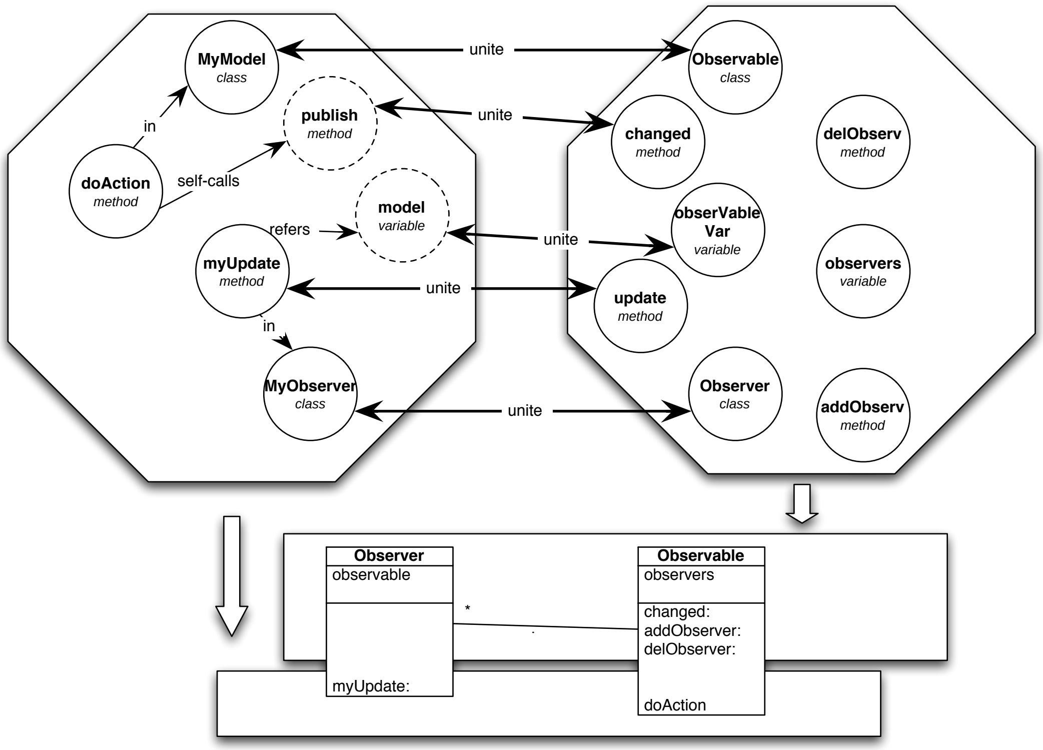


Figure 7.8: Observer generator in an integrative composition that results in an aspectual integration.

Combination of Aspectual Integration and Framework Specialization We can also specify an integrative composition that results in a hybrid combination of the two previous integrations. In the example integration in figure 7.9, we illustrate that the concrete observers are integrated with the generated observer implementation through subclassing, while the concrete observable implementation is integrated into the observable class.

Integration of Observer and Observable In particular cases, we might even want to integrate the observer and observable classes themselves. This is illustrated in figure 7.10. This is useful if the notification mechanism implemented on the

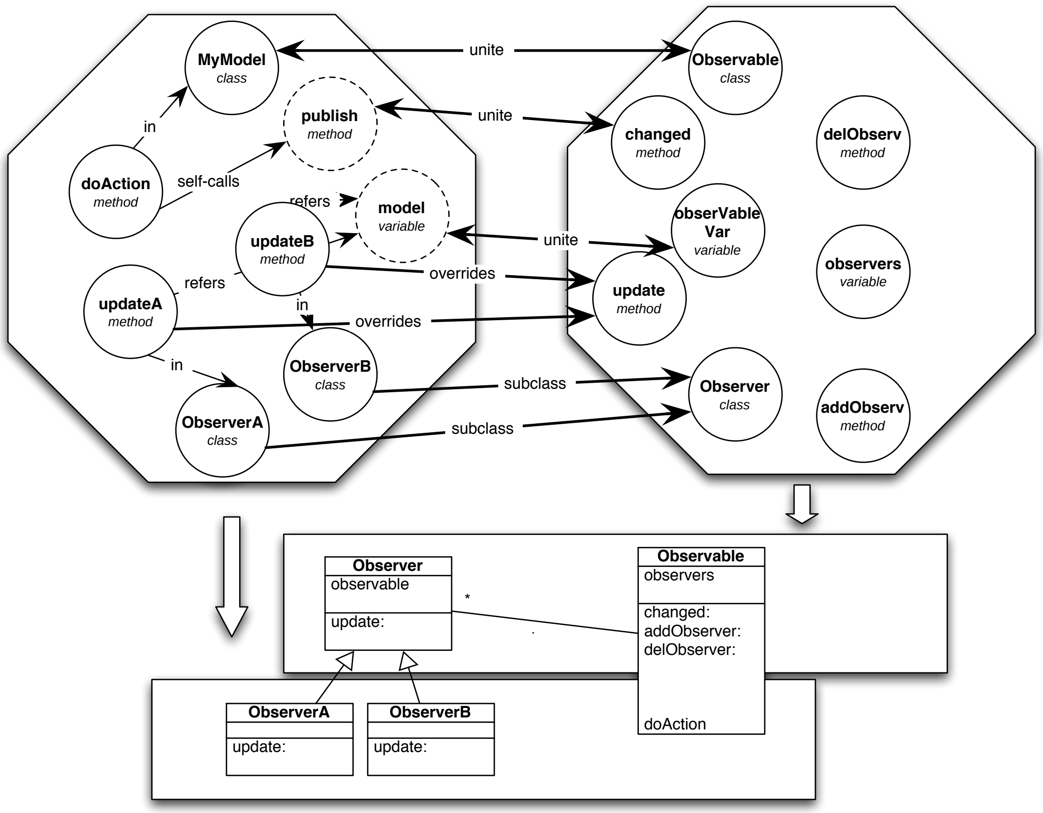


Figure 7.9: Integrative Composition that realizes a hybrid combination of an aspectual integration and framework specialization.

observable needs to propagate changes to itself. In fact, this can be seen as an aspectual integration of the observer and observable.

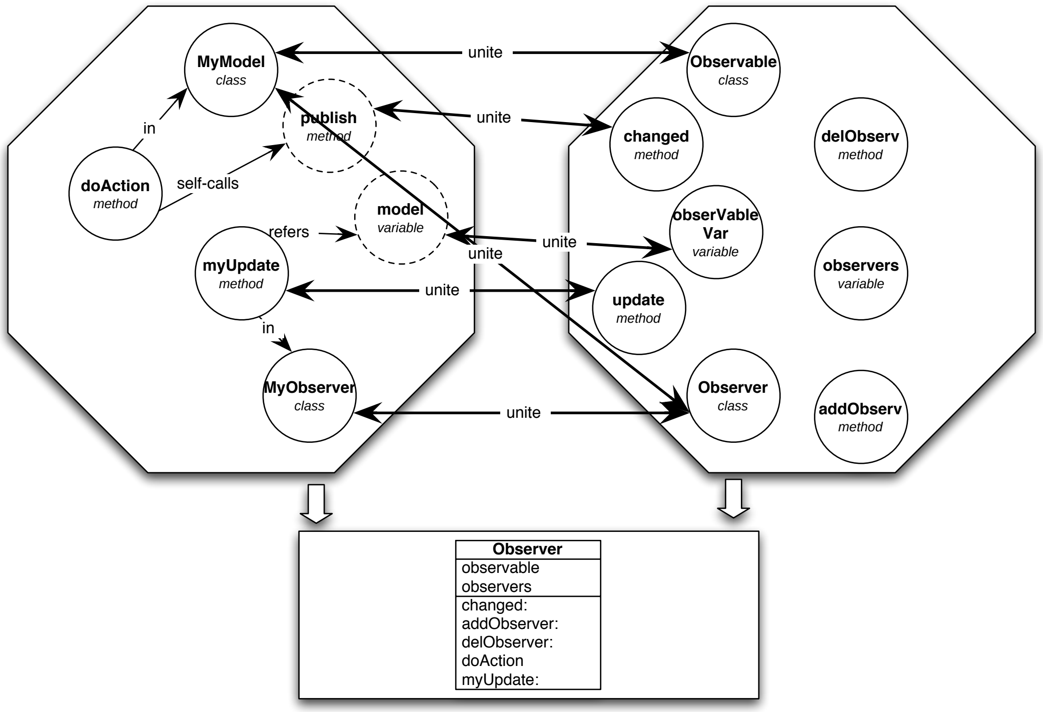


Figure 7.10: Integrative Composition of the Observer and Observable.

Integration of the observers variable The generated observer implementation also anticipates an integration of its `observers` variable with another variable through the `unite` integration relation. In that case, the types of the variables must match. The generated observer implementation uses a `Set` or an `OrderedCollection` in the `observers` variable, depending on the chosen features for the `notification` feature (i.e. `single` and `multi` respectively). The observer generator can adapt its generated program to use an `OrderedCollection` even if the `single` feature was chosen. The details of this adaptation can be found in the generation code for the `addObserv` method part, included in appendix B.

Other Integrations We can think of many more useful integrations with the generated observer-observable implementation. Some integrations might actually require the observer and observable to be subclasses of other classes and the observer and observable classes are open to integration of any kind of methods and variables.

7.3.2 Internal Implementation

The integrative compositions that were introduced in the previous section were possible because each generative program (of the separate program parts) has anticipated the possible integrative relations and composition conflicts. However, the alternative implementations and anticipations to integration relations in the observer generator were never targeted at specific integrative compositions with specific generators. Instead, each alternative implementation is driven by the predefined list of integration relations and composition conflicts in the Smalltalk language. In other words, the internal implementation deals with the integrative variabilities and determines the integrative commonalities. We illustrate this with some of the generative programs that actually implement the generation of specific parts of the observer-observable collaboration. A complete implementation of the observer generator is given in appendix B.

Generation of the Observer and Observable classes

The `Observer` and `Observable` classes anticipate an integration with other classes, either through subclassing or by a `unite` integration relation. We show the generative program for the `Observer` class in figure 7.11. It shows the three alternative implementations that can be generated. The first logic rule is the 'standard' class definition that is produced if no integration relation is applied to the `Observer` part. The second rule shows that the `Observer` part can adapt its class definition in case it is enforced by a `subclass` relationship to integrate as a subclass of another class. The third rule is applicable if a `unite` relationship is applied to the `Observer` part. In that case, the class definition can adapt its class name to the name of the class with which it is united. The implementation for the `Observable` part is almost identical. All three logic rules automatically anticipate to name conflicts because they call the `observerName/1` predicate. The implementation of this predicate is also included in figure 7.11 and shows three alternative names for the `Observer` class. Although we could specify a logic rule here that computes an infinite list of possible names, the specification of these names through facts allows us to define decent alternative names for the class. This often useful because the name of the class or method can be a part of the public interface of the generated program and can thus be called by manually written code.

```
Observer(class(?observerName,Object)) if
   observerName(?observerName).
Observer(class(?observerName,?superName)) if
   observerName(?observerName),
   subclass(Observer,?,class(?superName,?)).
Observer(class(?name,?super)) if
   unite(Observer,?,class(?name,?super)).

observerName(Observer).
observerName(AbstractObserver)
observerName(SuperObserver).
```

Figure 7.11: The Generative Logic Metaprogram for the `Observer` class part.

Generation of the Notification Methods

The `update` and `changed` program parts (produced by the observer generator) implement the methods that are the core of the observer-observable collaboration. The second method is called by an observable when its state changes. In its execution, the `changed` method program part invokes the `update` methods on the registered observers contained in the `observers` variable. The generative logic metaprogram in figure 7.12 implements the generation of the `update` program part. It consists of four alternative logic declarations. The first two logic rules implement the case where the feature `Parameter` was not included in the input specification. The last two rules implement the case where it is included. This can be noticed by the call to the `feature` predicate, which is also part of the generator's implementation. It is true if the featural description that is provided in the input specification of the generator contains the feature that is passed as an argument. The first and third logic rules generate the 'standard' implementation for the `update` method. The second and fourth rule implement the case where the method is involved in a `unite` integration relation. Mind that in both rules, it is verified if the method has the desired signature, i.e. a single parameter and two parameters respectively. This is done using the `singleKeyword` and `doubleKeyword` predicates that verify if a Smalltalk selector consists of one or two keywords respectively[3]. Otherwise, the generated program would be inconsistent with the desired features and a call from the `changed` method would result in an error.

The generative program for the `changed` method is included in figure 7.13. The

[3]In Smalltalk, the number of arguments of a method corresponds to the number of keywords in its selector (in case of a keyword message).

```
update(method(?class,{update:},{update:  anAspect ...},<>)) if
   not(feature(Parameter)),
   in(update,?,class(?class,?)).
update(method(?class,?selector,?body,?i)) if
   not(feature(Parameter)),
   in(update,?,class(?class,?)),
   unite(update,?,method(?,?selector,?body,?i)),
   singleKeyword(?selector).

update(method(?class,{update:from:},{update:asp from:anObservable...},<>)) if
   feature(Parameter),
   in(update,?,class(?class,?)).
update(method(?class,?selector,?body,?i)) if
   feature(Parameter),
   in(update,?,class(?class,?)),
   unite(update,?,method(?,?selector,?body,?i)),
   doubleKeyword(?selector).
```

Figure 7.12: The Generative Logic Metaprogram for the `Update` method part.

first rule implements the actual generation of the method. It automatically anticipates an integration in another class. If no integrative composition is made, the `in` predicate will return the class definition provided by the `Observable` class part, because the `changed` method part is involved in an `in` dependency relation with it. However, if an `in` integration relation is applied to the `changed` method part, then this predicate will return an alternative implementation for the class definition, i.e. the class defined by the part with which it is involved in this integration relation. Consequently, the `changed` method will automatically return an alternative implementation that fulfills the integration relation. Furthermore, the `Observable` class part is automatically integrated using a `unite` integration relation with the appropriate program part by the integration relation propagation mechanism. The generative program also automatically anticipates a different variable name for the `observers` variable part that contains the observers list. This is because a call to the `refers/3` predicate will subsequently result in alternative implementations for the variable. This also results in a different generated implementation for the `changed` method such that it fulfills the `refers` dependency relation with the `observers` part.

Finally, this generative program does not implement an anticipation to a `unite` integration or a name conflict. In the event of such a conflict or integration relation, it is the other program part (with which it conflicts or integrates) that is required to adapt. Mind that it would not be difficult to change the rule such that it anticipates name conflicts and `unite` integration relations. We merely illustrate here that it is not always required to do so, with the possible drawback that if the other generator does not anticipate a conflict or integration relation either, that the integrative composition will most likely fail.

```
changed(method(?class,{changed:},{changed:anAspect ?observers do:
                                        [:observer | observer ?updateMessage]},
                              <?observers,?updateMessage>)) if
   in(changed,?,class(?class,?)),
   refers(changed,observers,var(?class,?observers)),
   calls(changed,update,method(?,?updateSelector,?,?)),
   updateMessage(?updateSelector,?updateMessage).

updateMessage(?updateSelector,{?updateSelector anAspect}) if
   not(feature(parameter)).
updateMessage(?updateSelector,?updateMessage) if
   feature(parameter),
   makeMessage(?updateSelector,<anAspect,self>,?updateMessage)
```

Figure 7.13: The Generative Logic Metaprogram for the `Changed` method part.

7.4 User Interface Generator

The `User Interface` generator produces a class that implements all behavior of a user-interface (UI). The use of a dedicated user-interface modeling language is desirable and even necessary for an integration of separately generated user-interfaces. An integration of generated user interfaces at the Smalltalk level would be almost impossible to achieve without profound knowledge on the internal implementation of the generated user-interface implementations. For the purpose of this dissertation, we did not implement an entire user-interface modeling language, neither did we implement a complete user-interface generator. Instead, we implemented this generator on top of the existing user-interface generator in the Smalltalk Visualworks environment. The implementation of the generator thus boils down to the definition of a domain-specific user-interface language (defined on top of the Smalltalk UI generator). This language allows the modeling of windows, tabs, forms, textfields and menus. Of course, many more items could be included in a specification of an entire user interface. However, this is out of the scope of this dissertation. An overview of the UI language in its representational mapping in GLMP is included in table 7.1. These specifications are typically produced by a visual tool. Mind that we keep the specifications really simple here. We only include the necessary representation details useful for our examples.

UI Program Parts

Each of the possible UI specification parts contains an identifier that uniquely identifies the UI element (in `?id`). A `window` program part further contains the window title, the horizontal and vertical sizes and a `?parentWindow` identifier. If the window is a toplevel window, the identifier `System` must be used. A `menu` declares which window it is a menu for and a `menuItem` declares a name and an action which is a message that is sent when the menu is clicked. A `menuItem` also contains the identifier of the menu it is part of (`?menuId`). All following program parts are UI elements that are placed inside a window canvas. All of these program parts must include the coordinates of an upper-left corner (`?x1,?y1`) and a lower-right corner (`x2,y2`). These coordinates are the relative coordinates to the upper-left corner of the window canvas. There are three such UI elements: tab windows, input forms and text fields. The `tab` program part represents a tab window. An `inputForm` fact represents input forms and declares an accessor method, to be implemented on the generated UI class, which returns the value entered in the form. The `?changeEvent` must contain the name of message that is sent when the user of the editor changes the contents of the input form. Finally, a `textField` describes a name for the accessor that can be invoked to retrieve its contents. An example specification of a concrete user-interface is shown in figure 7.14.

UI specification part
`window(?id,?title,?horSize,?vertSize,?parentWindowId)`
`menu(?id,?parentWindowId)`
`menuItem(?id,?name,?action,?menuId)`
`tab(?id,?x1,?y1,?x2,?y2,?parentWindowId)`
`inputForm(?id,?accessor,?changeEvent,?x1,?y1,?x2,?y2,?parentWindowId)`
`textField(?id,?accessor,?x1,?y1,?x2,?y2,?parentWindowId)`

Table 7.1: The UI specification language.

The generator (depicted in figure 7.15) produces a class containing the necessary methods that are invoked by the Smalltalk system to open a user-interface (i.e. `viewSpec` and `open`). An appropriate set of instance variables together with accessors and mutators is included in this class for each of the values that can be displayed and/or edited in the UI. We will use this generator in the following section to specify domain-specific integrative compositions in the UI language. For that purpose, the

```
window(editWindow,{The Forms Editor},100,200,System).
inputForm(subjectForm,subject,subjectChanged,5,5,100,45,editWindow).
inputForm(authorForm,,author,authorChanged,5,55,100,95,editWindow).
inputForm(accountForm,,account,accountChanged,5,105,100,150, editWindow).
inputForm(detailsForm,details,detailsChanged,5,155,100,195, editWindow).
```

Figure 7.14: An example user-interface specification.

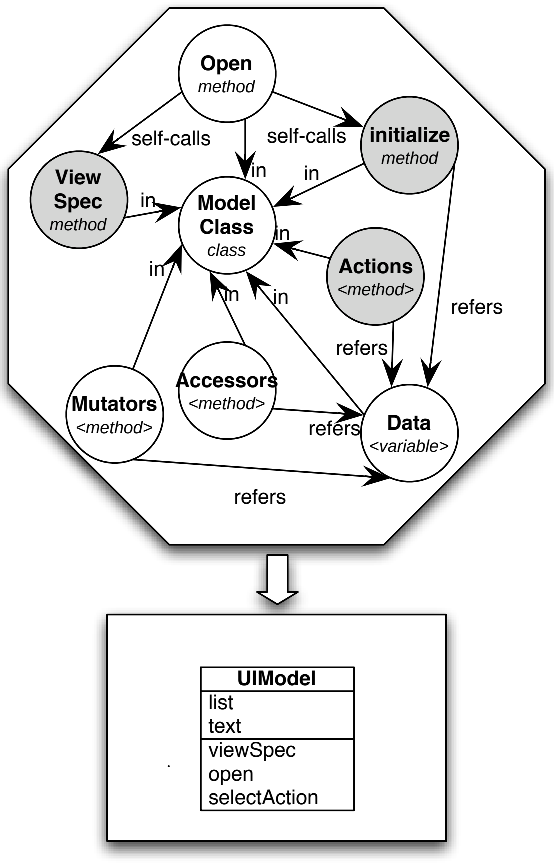


Figure 7.15: The User Interface Generator with a sample generated program

UI language is equiped with the integration and dependency relations that are described in table 7.2. All relations can be used as integration relations, but only the `in`, `of`,`left` and `above` relations can be used as dependency relations. More complex integration relations were not considered and require more research about desirable user-interface integrations. There is also one integration relation propagation rule, depicted in figure 7.16. The propagation rule implements that menu's need to be united if one of their menu-items is united.

Relation	**Description**
`in(?element,?window)`	The `?element` program part is integrated in the `?window`. This means that the element becomes embedded in the ?window
`of(?menuItem,?menu)`	The `?menuItem` part is included as an menu item of the `?menu`.
`left(?itemA,?itemB)`	The `?itemA` needs to be placed left of `?itemB`.
`above(?itemA,?itemB)`	The `?itemA` needs to be placed above `?itemB`.
`unite(?menuA,?menuB)`	The `?menuA` part is united with `?menuB`. Their menu items are grouped.
`unite(?menuItemA,?menuItemB)`	The `?menuItemA` part is united with `?menuItemB`.

Table 7.2: User-interface integration relations.

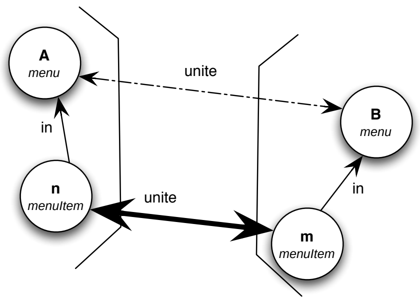


Figure 7.16: User-interface integration propagation rules.

Composition Conflicts

The most important composition conflicts that we deal with in the UI language are *overlapping UI elements*. Therefore, the UI language implements a constraint that is automatically imposed between all UI elements placed in the same window. This constraint verifies if the coordinates defined by the UI element program parts do not result in overlapping UI elements. Other composition conflicts of the UI language are about inconsistent UI descriptions, such as multiple systemwindows, identical menu item names, ...

7.5 Composed Generators

Until now, we have described some atomic generators that are included in the library of program generators. We will now discuss some composed generators and generators that produce an implementation that needs to integrate with the generated programs of other generators.

7.5.1 Consistent Data Containers

The use of multiple instances of a data container in an application can require that their contents remain consistent. This is specifically the case when two different kinds of data containers need to contain the same elements. For example one data container may be a plain set containing objects, while the other data container is a dictionary that stores key-value pairs. In this way, the same data is available in different data containers that are each appropriate for a particular kind of use. For example, searching for particular objects, given a key, is much faster in a dictionary than in a set. Conversely, a set offers a quick iteration over all contained objects. Consequently, in a single software application, two different kinds of generated data containers can exist while their contents need to be the same. The implementation of the consistency mechanism, that keeps the contents of the data containers consistent, is produced by a separate generator that is composed with the data container generators. The implementation of a separate consistency generator is necessary because it produces a collaboration between two generated data containers and a data container generator produces only a single data container.

Implementation

The generator produces an implementation that is able to synchronize multiple instances of both kinds of data containers. The consistency generator and a sample generated program are shown in figure 7.17. The generated program consists of additional methods and variables that need to be added to the data container classes.

Each data container implementation class needs to be extended with a `syncWith:` method. This method can be invoked on a data container instance to keep it consistent with another data container instance (which is given as argument). Each data container also needs to keep a reference to all other instances that need to be kept consistent with it. Therefore a `containers` instance variable is introduced on each data container class. The actual consistency-keeping behavior is implemented in the appropriate generated methods that need to be combined with the addition and removal methods of the data container implementation. In the figure, these methods are shown in italic (e.g. `sendAdd:` and `sendRemove:`). This is because they need to be combined with the actual addition and removal methods produced by the data container generators. Furthermore, to prevent an infinite loop in the update mechanism, the `block` variable is used to keep the state of the consistency implementation: i.e. busy updating or not.

The consistency generator requires the featural descriptions of the data containers that need to be integrated. It also accepts a specification on how to convert values of one container into values of the other container. This specification consists of messages that can be sent to the values to convert them. An example specification that declares that the messages `asB` and `asA` need to be sent to the values to convert them from a value for one container to a value for the other container is shown below. It also includes the specification of a message that can be sent to retrieve a key from a value. This is required because the generated consistency implementation can keep the contents of data containers that are configured with a different `Indexing` feature (e.g. `plain` and `keyvalue`) consistent. Therefore, the consistency generator requires that its input specification specifies the message that can be sent to a value object to calculate its key.

```
convertAtoB({asB}).  convertBtoA({asA}).  retrieve({asKey})
```

As shown in figure 7.17, the generator that produces the consistency book-keeping code is built as a translation composition of two generators. The bottom generator produces the actual Smalltalk code. The top generator is situated at the feature description level and produces a featural description of the code that needs to be generated. This description simply consists of the featural descriptions of the data containers involved in the consistency implementation. In an integrative composition, the consistency generator does not only require integration relations to operate on its Smalltalk program parts but also on the featural descriptions. This is necessary for two reasons. First of all, the generated consistency implementation must differ depending on the featural descriptions of the data containers. For example, depending on the chosen feature for `Size`, the consistency implementation must or must not contain a check to verify if a data container is full or not. The second reason is that the

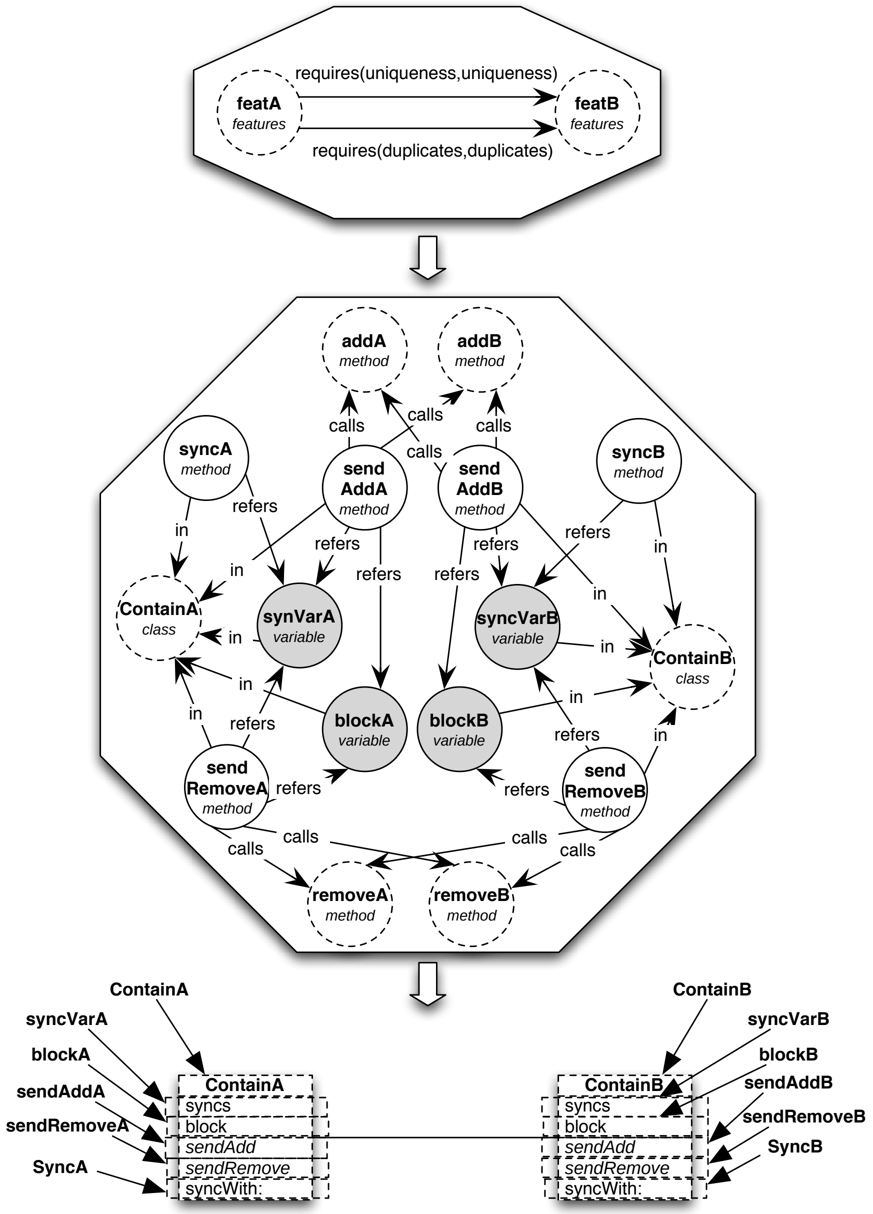


Figure 7.17: The implementation of the Consistency Generator and a sample generated program.

consistency generator can only produce a consistency-keeping implementation if the featural descriptions of both data containers are compatible with respect to *data management* and *indexing*. This is because it is impossible to keep data containers with duplicates consistent with a data container without duplicates. The complete implementation of the consistency generator is included in appendix C.

Integrative Composition

An integrative composition of two data container generators and the consistency generator is illustrated in figure 7.18. The necessary integration relations are drawn and the data container implementation is automatically adapted to accommodate the integration of the consistency implementation. This involves both an integration at the Smalltalk level and at the FDL level. At the Smalltalk level, we need to integrate the required program parts of the consistency generator with the appropriate parts of the data container generators using `unite` relations. We also need to declare the `includeAfter` integration relation between the addition and removal methods of the program generators. At the FDL level, we simply need to provide the consistency generator with the featural descriptions of both data containers. This merely requires an integration with the required FDL program parts using a `unite`. In the composition, the consistency generator enforces the dependency relations that are shown between the featural descriptions of both generators. If these dependencies are violated, the composition cannot occur.

Integrative Composability

The `Consistency` generator can integrate two different data containers but still puts some requirements on the featural descriptions of both data containers. If these requirements are not met, then the consistency generator cannot produce an implementation and the integrative composition fails. However, if there are alternative featural descriptions for the data containers, then the integrative composition can work through the (automatic) selection of such an alternative. There can exist alternative featural descriptions if the featural descriptions are produced by another generator and thus, if the data container generator is in fact part of a composed generator. Such an example is described in the next subsection.

7.5.2 Document Editor Generator

A document editor is one of the tools produced by the company. Clients have a variety of requirements when they order a document editor. These requirements relate to features such as lay out of a document, syntax coloring, user-interface, storage of the document, etc.... Since the company produces different document editors for many

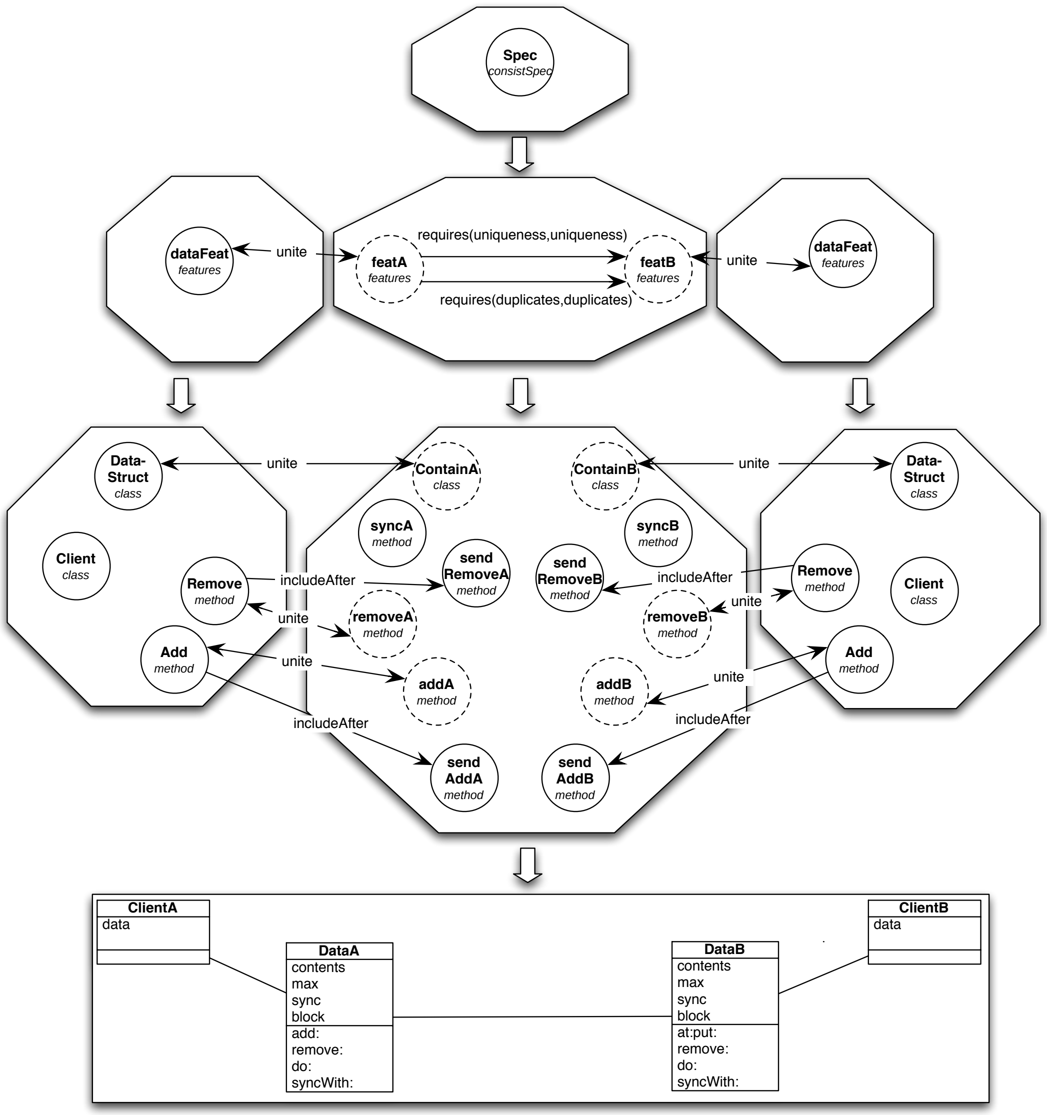


Figure 7.18: Composition of the Data Container and Consistency generators.

different clients, it pays off to develop a program generator for document editors that represents a product-line of document editors that vary in the aforementioned features. The developers alleviate the development of this generator by reusing the generators of the library we have introduced above. Consequently, the developers implement the document editor generator as an integrative and translation composition of other generators. We do not elaborate on a feature diagram and possible input specifications for this generator. Instead, we focus on the use of integrative composition to construct this generator.

Figure 7.19 illustrates the composition of generators in the implementation of the document editor generator. The client's requirements are expressed using a featural description in the `Document Editor FDL`. These descriptions are accepted by the `DocEditComp` generator. This generator translates these `Document Editor FDL` featural descriptions into `Data Container FDL` featural descriptions and UI specs for the `Data Container` and `User Interface` generators respectively. Furthermore, it also produces a specification for the `SubDocEdit` generator. Both the `SubDocEdit` and `DocEditComp` generators are specific to the implementation of the `Document Editor` generator.

DocEditComp Generator

The `DocEditComp` generator produces a *consistent* set of input specifications for the `SubDocEdit`,`User Interface` and `Data Container` generators. This means that it produces an input specification for each of the other program generators in the composition such that they can interoperate. In essence, the `DocEditComp` generator encodes the dependencies between the featural descriptions of each of the generated program parts. A part of this is shown in figure 7.20. This figure shows the `DocEditComp` generator and visualizes the dependencies between the featural description of the data container structure and the featural description that is accepted by the `SubDocEdit` generator. It shows two possible alternative featural descriptions for each program part. The dependency states that when the featural description of the data container contains the `Duplicates` feature, then the `autoDuplicates` feature is required in the featural description provided to the `SubDocEdit` generator. This is because the code of the generated document editor needs to be different when a data container can contain duplicates or not. The `DocEditComp` generator produces two alternative configurations for a data container because it anticipates a composition problem when the data container is integrated with another data container (e.g. through the `Consistency` generator, which is explained later on).

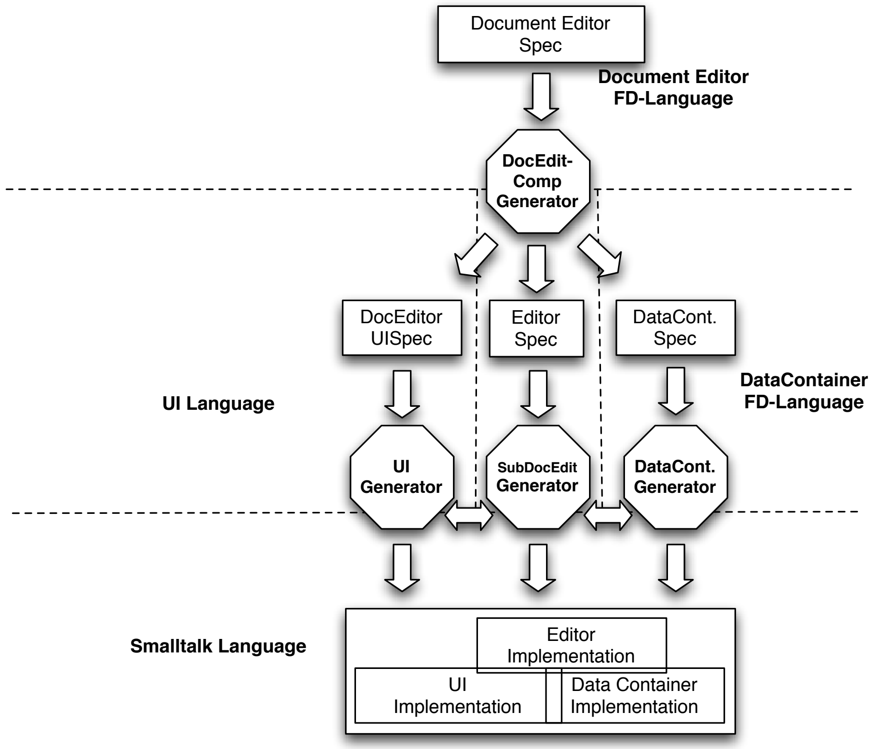


Figure 7.19: Schematic overview of the compositions in the Document Editor Generator.

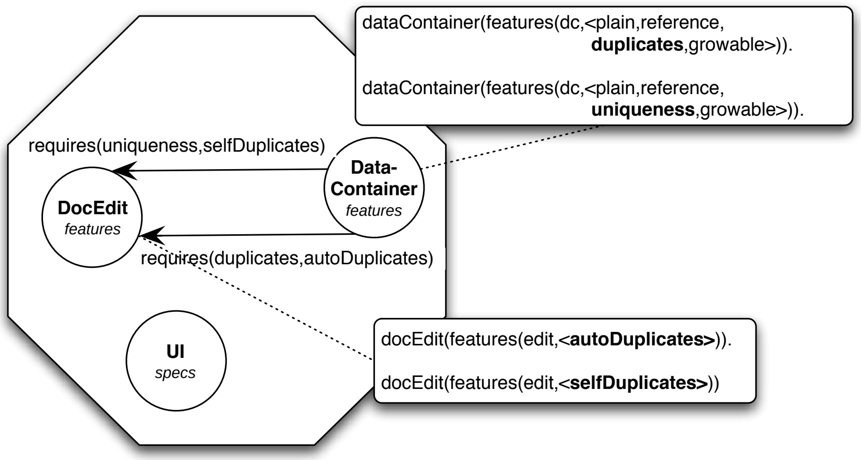

dataContainer(features(dc,<plain,reference, **duplicates**,growable>)).
dataContainer(features(dc,<plain,reference, **uniqueness**,growable>)).
requires(uniqueness,selfDuplicates)
Data-Container
features
DocEdit
features
requires(duplicates,autoDuplicates)
UI
specs
docEdit(features(edit,<**autoDuplicates>**)).
docEdit(features(edit,<**selfDuplicates>**))

Figure 7.20: Partial DocEditComp Generator.

SubDocEdit Generator

The `SubDocEdit` generator produces the Smalltalk implementation of the document editor that integrates with the generated data container and the generated user-interface. In this experiment, we did not completely implement this generator. We only implemented those parts that were necessary to validate the integrative composition technique. In essence, this consists of a class with the methods that need to interact with the generated user-interface and data container implementations. Its integrative composition interface is described in the context of the integrative composition in the following paragraph.

Integrative Composition

The integrative compositions required to construct the `Document Editor` generator are simple because they are not invasive. The integrative composition is shown in figure 7.21. The UI generator produces a class with methods that are called by the generated document editor implementation. Similarly, the document editor implementation calls the methods of the generated data container. The only interactions between the generated implementations are method calls. Therefore, the integration simply requires that the `SubDocEdit` generator is supplied with the names of the classes and methods that need to be called. This is achieved through `unite` integration relations between the necessary program parts. The `SubDocEdit` generator is specifically built for an integrative composition with the `Data Container` and `User Interface` generators and consequently exposes the necessary required program parts in its composition interface.

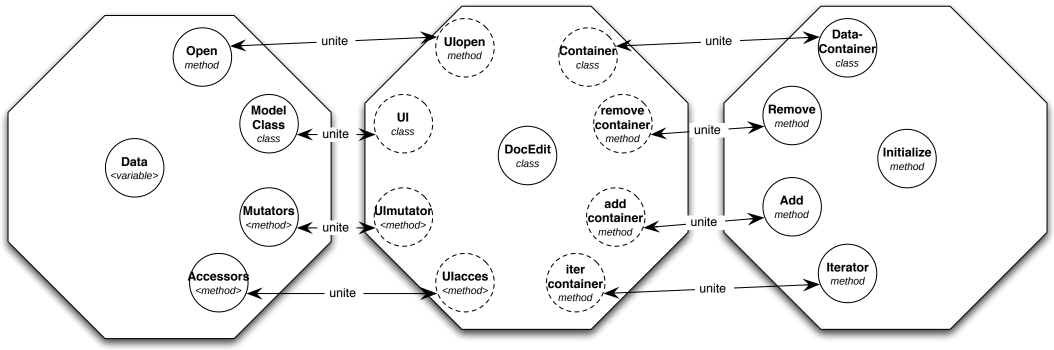


Figure 7.21: The integrative composition of the `User Interface` generator (left), the `SubDocEdit` generator (center) and the `Data Container` generator (right).

7.5.3 Forms Editor Generator

The `Forms Editor` generator produces a tool that is similar to the document editor. However, the forms editor has a user-interface that consists of forms that need to be filled in. Given these forms, it produces a document. In other words, the forms editor provides a limited view on a document, while the document editor allows full word processing abilities. The composition structure of the `Forms Editor` generator is identical to the composition structure of the `Document Editor` generator. It also contains the a `Data Container` and `User Interface` generators that are composed with a specific generator that produces the functionality of the forms editor as well as a generator that translates the featural description of the forms editor into a consistent set of featural descriptions of the data container, user-interface and forms editor.

7.5.4 Integrated Document Editor Generator

The `Integrated Document Editor` generator is built as a composition of the previous two generators: the `Forms Editor` generator and the `Document Editor` generator. The composition scheme to build this generator is shown in figure 7.22. Since both generators are composed generators, the composition interface not only consists of the generated Smalltalk program parts but also of the intermediate generated programs.

These intermediate programs are the featural descriptions and input specifications for, amongst others, the `Data Container` and `User Interface` generators. The integrative composition of the `Document Editor` and `Forms Editor` generators is expressed in terms of the user-interface specification and includes the composition of the `Consistency` generator. We describe these two aspects of the integrative composition: the integrative composition of the user-interfaces and the integrative composition of the data containers using the consistency generator.

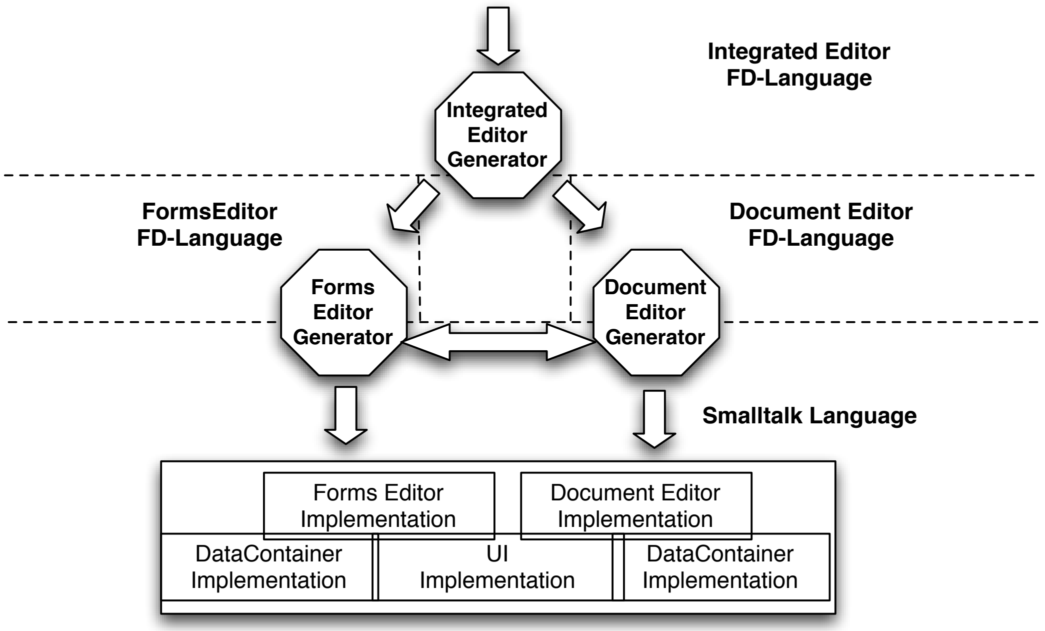

Integrated Editor FD-Language
Integrated Editor Generator
FormsEditor FD-Language
Document Editor FD-Language
Forms Editor Generator
Document Editor Generator
Smalltalk Language
Forms Editor Implementation
Document Editor Implementation
DataContainer Implementation
UI Implementation
DataContainer Implementation

Figure 7.22: Schematic overview of the composition to create the Integrated Document Editor.

Consistency Generator Composition

We first consider the integrative composition of the data containers using the `Consistency` generator. The `Document Editor` generator produces a data container that can contain duplicates, while the `Form Editor` generator produces a data container that does not contain duplicates. However, both need to be kept consistent through the code produced by the `Consistency` generator. This would normally lead to a composition conflict. Fortunately, the developer of the `Document Editor` generator anticipated the possible composition conflicts that can occur with the featural description of the data container and provided an alternative implementation that allows the use of a datacontainer without duplicates. This composition conflict can consequently be automatically resolved by the generative system. The essential part of the integrative composition is shown in figure 7.23. The details of this integrative composition are identical to the integrative composition visualized earlier on, in figure 7.18.

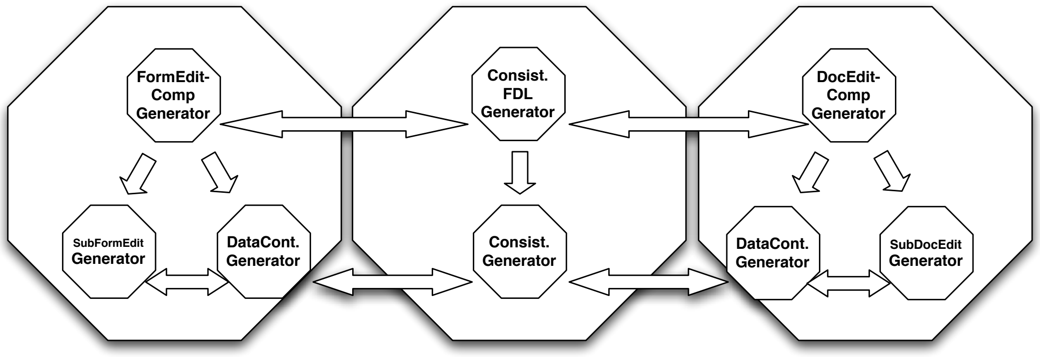


Figure 7.23: Focus on the composition of the `Consistency` Generator in the `Integrated Document Editor` generator.

UI Composition

The following integrative composition is the domain-specific integrative composition of the user interfaces. An integrated document editor is an application that integrates the forms editor and document editor applications in a single application. Consequently, the user-interfaces need to be integrated. However, these user-interfaces are produced by separate generators (the `DocEditComp` and `FormEditComp` generators). We can specify an integration at the UI-specification level because the integrative composition interfaces of the `Document Editor` and `Forms Editor` generators expose the generated program parts of the UI specification. Table 7.3 shows the program parts and the generative programs for the forms editor UI specification in the `FormEditComp` generator. Likewise, table 7.4 shows those of the document editor UI specification, implemented in the `DocEditComp` generator. They show how the specification of the windows can adapt to the different integration relations that can be applied to them. We will clarify them in the description of two different UI integrations:

Tabwindow We can integrate both windows in a common user-interface by placing the separate windows in a tabwindow. Therefore, the `Integrated Editor` generator produces a specification of a user-interface that consists of a single tab window. Next, we specify the integrative composition of these three generators (shown below). The composed generator produces a single UI specification in which both the document and forms editor windows are integrated into a tabwindow. In tables 7.3 and 7.4, we can see that the generative programs for the `formsEditWindow` and `docEditWindow` program parts anticipate the possible integration in a tabwindow. If they are integrated in a tabwindow, the title and sizes of the window are adapted. In a tabwindow, the titles of the tabs are the titles of the windows that are integrated in them. The integration relations for this integrative composition are:

```
in(docEditWindow,integratedTabWindow).
in(formsEditWindow,integratedTabWindow).
unite(formsMenu,documentMenu).
```

Merge Another integrative composition can be specified that results in the merge of both the document and forms editor windows in a single window. This requires that the individual UI elements produced by the different generators are correctly placed (i.e. not overlapping) in the single window. This is ensured by the composition conflict detection mechanism and is anticipated in the implementation of both generators. The `textField` program part either occupies the entire window or it occupies only half of the window. Next, the `forms` program listpart invokes a separately defined predicate that provides many alternative distributions of the forms over the entire surface. If possible, a distribution of the forms that still allows the placement of the text field will be found and the integration will succeed. Of course, this second integrative composition requires a different implementation of the `Integrated Editor` generator (i.e. where the tab window is left out). The integration relations for this integrative composition are:

```
unite(docEditWindow, formsEditWindow).
unite(docEditWindow,integratedWindow).
```

Figure 7.24 schematically illustrates the required composition of all generators involved (except for the consistency generator). It shows how the generative system

Program Part	**Generative Program**
formsEdit- Window	formsEditWindow(window(formsEditWindow,{The Forms Editor},200,200,System)). formsEditWindow(window(formsEditWindow,{Input Forms Tab},?xSize,?ySize,?id)) if in(formsEditWindow,?,tab(?id,?x1,?y1,?x2,?y2,?)), minus(?x2,?x1,?xSize), minus(?y2,?y1,?ySize). formsEditWindow(?window) if unite(formsEditWindow,?,?window).
formsEdit- Menu	formsEditMenu(menu(formsEditMenu,?id)) if of(formsEditMenu,?,?id).

forms	forms(?forms) if
	 in(forms,?,window(?id,?,?xSize,?ySize,?)),
	 distributeForms(?xSize,?ySize,
	   <inputForm(subjectForm,subject,subjectChanged,?id),
	   inputForm(authorForm,author,authorChanged,?id),
	   inputForm(accountForm,account,accountChanged,?id),
	   inputForm(detailsForm,details,detailsChanged,?id)>,
	   ?forms).

Table 7.3: Forms Editor generator implementation

Program Part	**Generative Program**
docEdit- Window	formsEditWindow(window(docEditWindow,{The Document Editor},200,200,System)). docEditWindow(window(docEditWindow,{Document Tab},?xSize,?ySize,?id)) if in(docEditWindow,?,tab(?id,?x1,?y1,?x2,?y2,?)), minus(?x2,?x1,?xSize), minus(?y2,?y1,?ySize). docEditWindow(?window) if unite(docEditWindow,?,?window).
docEdit- Menu	docEditMenu(menu(docEditWindow,?id)) if of(docEditMenu, docEditWindow,window(?id,?,?)).
docEditText	docEditText(textField(docEditText,text,1,1,?xSize,?ySize,?id)) if in(docEditText,?,window(?id,?,?xSize,?ySize,?)). docEditText(textField(docEditText,text,1,y1,?xSize,?y2,?id)) if in(docEditText, docEditWindow,window(?id,?,?xSize,?ySize,?)), quotient(?ySize,2,?yHalf), interval(?y1,1,?yHalf), interval(?y2,?yHalf,?ySize).

Table 7.4: Document Editor generator implementation

Program Part	**Generative Program**
integrated- Window	integratedWindow(window(integratedWindow,{The Integrated Editor},300,300,System)).
integrated- Tabwindow	integratedTabWindow(tab(integratedTabWindow,1,1,?xSize,?ySize,?id)) if in(integratedTabWindow,integratedWindow,window(?id,?,?xSize,?ySize,?))

Table 7.5: Integrated Editor generator (UI generation part).

composes the generators of both the `Document Editor` and `Forms Editor` generators. For clarity, it does not illustrate the integrative composition of the consistency generator with the data container generators.

7.6 Discussion

The set of experiments presented in this chapter has shown that the GLMP technique provides a powerful integrative composition technique. We have explicitly described how atomic generators (e.g. the `Observer` and `Data Container` generators) can be involved in several different integrative compositions. The experiment with the `Observer` generator demonstrated specifically how a generated program can be integrated with another generated program in many different ways. None of these integrative compositions has to be anticipated as a whole. Instead, the developer of the generator needs to design the set of program parts and anticipate the integration relations that can be applied to each program part.

The integration of the `Data Container` and `Consistency` generators has shown how the FDL approach is useful to detect high-level composition conflicts without the need for defining a separate domain-specific language to detect them. The generated consistency-keeping implementation is also an example of crosscutting code. Crosscutting code requires an integrative composition technique to integrate it correctly with other generated programs. We described the integrative composition of the `Data Container` and `Consistency` generators in detail and afterwards we described their composition as a part of the `Integrated Document Editor` generator. In this final experiment, we demonstrated the ability to detect and resolve high-level composition conflicts in both the FDL and `User-Interface` languages. This experiment has specifically shown the appropriateness of domain-specific integrative compositions. We have shown how UI-specifications can be integrated and how the domain-specific conflict of overlapping UI elements can be detected and resolved.

The design and implementation of integrative composable generators also has an impact on other characteristics of program generators. We discuss some of these issues here.

Commonalities and Variabilities

The implementation technique for integrative composable generators impacts the commonalities and variabilities of the generated programs. All generated programs of a generator consist of the program parts defined by the generator. Furthermore, these program parts are always related by the dependency relations defined by the generator. Consequently, the commonalities of all generated programs of a single program generator are the program parts and their dependencies. The variabilities are in the

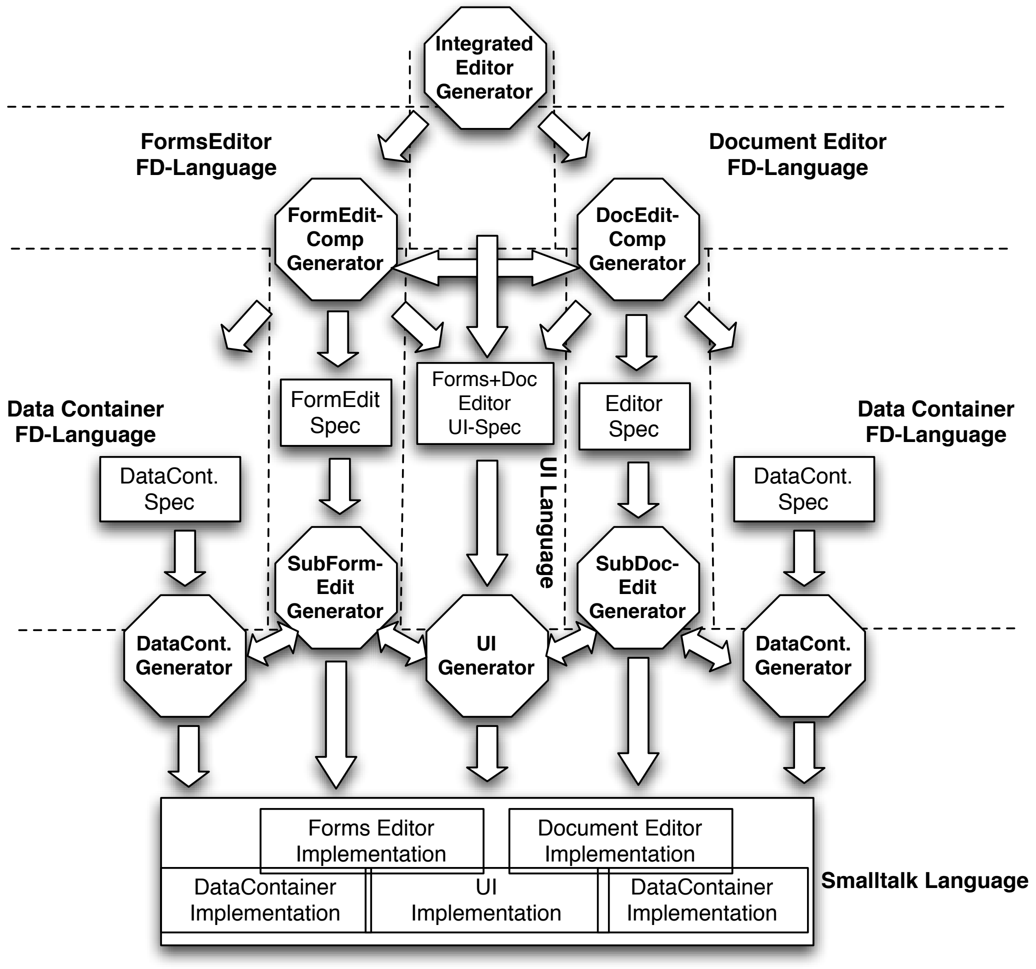


Figure 7.24: Detailed overview of the composition that creates the Integrated Document Editor.

implementation of each generated program part and in the integrations with other programs.

Scalability

Program generators are a scalable implementation for efficient and configurable reusable software artifacts. Although the implementation of a program generator is inherently more complex than the programs it generates, a generator pays off because its implementation represents an entire family of generated programs. Integrative composable generators even enhance the scalability because they represent an entire family of generated programs and each of these generated programs *can adapt to particular integrations* with other (generated) programs. In other words, the number of possible generated programs is increased because each generated program can also be adapted in the context of an integrative composition.

Of course, this improved scalability has an impact on the implementation complexity of the generator because the developer of the generator also needs to implement the integrative variabilities. In this dissertation, we have provided support to deal with these integrative variabilities through separate program parts, dependency relations and alternative generated programs. The complexity of each generative program grows with the number of dependencies and integration relations it can be involved in. Consequently, the more integrative compositions that are allowed by the program generators (i.e. the more public program parts), the more complex the implementation. GLMP allows to tackle this complexity because each generative program only deals with the generation of its own program part and thus has to focus only on the adaptations of its own program part. Furthermore, each generative program is automatically provided with all possible adaptations to the program parts it depends on via the dependency relations and the associated parameterization mechanism. The constraint checker automatically enforces that a correct set of adaptations for all program parts are chosen such that the generated program is correct.

Performance of Generation and Generated Programs

The intention of program generation is the production of an efficient implementation for a software artifact. This efficiency primarily stems from the fact that all statically configurable adaptations to the generated program are executed at generation-time, preventing unnecessary performance overhead at runtime. Our approach does not invalidate this goal. One could argue that the generated program often has to be structured such that integrative compositions are possible. This structure can conflict with a more efficient implementation of the generated program. This is true but although we did not investigate these issues, we expect these to be of minor importance with respect to the performance gained because of the program generation itself.

Furthermore, the performance of the GLMP system itself was also not an issue in this dissertation. Although the prototype implementation can be improved considerably, the execution of logic programs and the solving of constraint networks remains a slow compilation technique. Nevertheless, the generation of a program has to occur only during the development process.

7.7 Conclusion

We have validated the GLMP technique that is proposed in this dissertation by implementing several integrative composable program generators. For a number of these generators, we have shown how the generator can be involved in many different integrative compositions, that result in different integrations of their generated programs. We have specifically shown this for the observer-observable generator. For one particular generator (the UI generator), we defined a simple domain-specific modeling language such that integrations can be specified at the domain-specific level. Furthermore, we have detected domain-specific composition conflicts and we have shown how the implementation can anticipate and resolve them. However, because identifying, implementing and correctly using domain-specific modeling languages in the implementation of generators is a complex and tedious task, we have also used a modeling language (FDL) that can be used by all generators. We have shown how program generators that use this common modeling language are composed and how this allows to prevent high-level composition conflicts between generators implemented in a single library.

Chapter 8

Conclusion and Future Work

In this dissertation, we have supported the thesis that integrative composition of program generators is necessary for the modular implementation of program generators as a composition of other generators. In this chapter, we conclude the dissertation and summarize the major contributions. We also discuss the limitations of the proposed technique and mention future research directions.

8.1 Summary

In this dissertation, we have identified the need for *integrative composition* of program generators. An integrative composition of program generators results in a composed generator that produces an (invasive) integration of the programs generated by its constituents. Building program generators as an integrative composition of other generators provides us with the opportunity to modularize the implementation of a program generator according to the concerns that need to be generated. In essence, an integrative composition allows to compose generators, that each generate the implementation of a single concern, into a composed generator that produces a program that implements all concerns. However, the need for integrative composition conflicts with the encapsulation and black-box property of the generated programs. First of all, to specify an integrative composition, we need knowledge on some of the internal implementation details of the generated programs. Furthermore, many interferences might occur in an integration, which require manual adaptation of either the generator or the generated programs, which is most undesirable. To reconcile the advantages of black-box generated programs with the need for integrative composition, we have described a number of requirements that an integrative composition mechanism must adhere to. Summarized, these requirements state that integrative composable generators need to provide controlled access to parts of their generated program such that an unanticipated integration of their generated programs can be specified. Furthermore, the integrative composition mechanism needs to prevent inadvertent interferences in the integration of the generated programs by means of a conflict detection and resolution mechanism.

We introduced a generative programming technique that allows the building of *integrative composable generators* and that adheres to requirements that we set forth. We first introduced the main concepts of our technique, which are independently of a particular implementation technology. Afterwards, we provided a prototype implementation of the technique using generative logic metaprogramming. We demonstrated the building of integrative composable generators, as well as their integrative composition, using a number of example generators. We have shown how generators are equipped with an *integrative composition interface* that exposes parts of their generated programs. An actual integrative composition is specified by means of *integration relations* that are defined between the exposed parts in different composition interfaces. The integration relations do not only enforce a particular integration of the generated program parts, they also influence the generative programs to achieve the desired integration. Attention was drawn to the fact that the possible integration relations and kinds of parts that can be exposed in a composition interface are defined for each possible output language of a generator. This means that all program generators that produce a program in the same language can be involved in an integrative composition. Furthermore, we have shown how integrative composable generators are designed and implemented for integrative composition by a mechanism of *dependency relations* between program parts and *alternative generative programs* for each program part. These mechanisms allow the developer of a generator to deal with *integrative variabilities* which are necessary to anticipate and resolve the possible interferences that might occur in an integrative composition. The possible interferences themselves are declared for each language and are automatically detected in each integrative composition. We also described how the combination of translation composition and integrative composition provides opportunities to specify an integration at higher levels of abstraction, i.e. in *domain-specific (modeling) languages*. We argued that such domain-specific integrative compositions are more appropriate since they can specify the integration in more domain-specific terms and can be checked for *domain-specific composition conflicts*. We have presented examples of such domain-specific integrative compositions in the Tree and User Interface languages.

Although the design of appropriate domain-specific languages provides the opportunity for domain-specific integrative compositions, the identification, design and implementation of these domain-specific modeling languages and their use in the development of all generators is a tedious and difficult task. Moreover, their definition and consistent use is even impossible in an open environment. To enable the detection and resolution of domain-specific interferences without the need to develop a domain-specific modeling language, we have also presented a featural description language and demonstrated its use in the development of a library of integrative composable program generators. We have shown that an integrative composition of these program generators can lead to an alternative selection of features and the adaptation of the generated low-level programs according to those features.

8.2 Conclusion

Program generation is at the heart of recent software development paradigms such as generative programming [Cza98], product-line engineering [BJMvH02, BLHM02] and model-driven architectures (MDA) [Gro]. In these paradigms, program generators are used as a scalable implementation technique for an entire family of reusable program parts. A generator accepts a high-level description of the desired program part and generates a corresponding low-level implementation for it. Although program generation is a powerful implementation technique for reusable program parts, program generators themselves are hardly reusable in the implementation of other generators. This is because program generators are considered in isolation and are not designed nor implemented for *integrative composition.* Using contemporary generative programming techniques, an integration of the generated programs needs to be performed manually and often requires adaptations to the program generators or generated programs. These adaptations are necessary to achieve a correct invasive integration of the generated programs and to prevent undesired interferences that break the functionality of the generated programs. The need for such manual adaptations renders the modular reuse of contemporary program generators impossible.

In this dissertation, we proposed and implemented an integrative composition technique in which the required adaptations to the generated programs are performed by the program generators themselves. This is possible because we can anticipate the possible integrative variabilities during the implementation of a program generator and implement them appropriately. The implementation of these integrative variabilities must deal with the necessary adaptations caused by the possible integrations and interferences in the generated program, which are defined for a given implementation language. The integrative composition mechanism enforces the generators to produce an integrated program that adheres to the integration specification and does not contain defined undesired interferences. The technique of generative logic metaprogramming proved to be a powerful implementation technique for integrative composable generators. This is because the logic language provides appropriate linguistic support for the implementation of the integrative variabilities.

Integrative composable generators allow us to apply the principle of separation of concerns in the context of a modular implementation of program generators. We can structure the implementation of a program generator according to the concerns that need to implemented in the generated program. The generation of each concern can be encapsulated in an individual program generator and a composition of program generators produces a program that contains the (invasive) integration of all concerns. The implementation of each concern is appropriately adapted to implement the desired interactions and to prevent the undesired interferences.

8.2.1 Contributions

In this section, we summarize the major contributions of this thesis.

Modular Integrative Composition We have identified the need for integrative composition of program generators for the modular implementation of program generators as a composition of other generators. The presented mechanism for integrative composition establishes a correct integration of the generated programs without the need for extensive knowledge on the internal implementation of the generated programs. To perform an integrative composition of program generators, only the implementation details that are exposed through the integrative composition interface are required. The integrative composition mechanism ensures that the individual generated programs are integrated as specified without the defined and detectable interferences.

Dealing with Integrative Variabilities An integrative composition can only occur if the generators involved in the composition are able to integrate their generated programs. This means that the generated programs need to be adapted to achieve an (invasive) integration and to resolve possible interferences. We have identified these adaptations for integrative composition of the generated programs as integrative variabilities. The generative technique that was developed in this dissertation allows to explicitly deal with these integrative variabilities in the implementation of a program generator. The fundamental mechanism for this is the generation of alternative implementations for individual program parts of the generated program. This allows to tackle the integrative variabilities through alternative implementations only where they are needed. The mechanism of dependency relations and integration relation propagation ensures that the entire generated program is consistent with the chosen alternative implementation of a particular part. In other words, the dependency relations and integration propagations ensure that the entire generated program is adapted accordingly because of an adaptation to a single program part. The choice of an alternative implementation is driven by the possible interferences and the possible integrations that can be specified in the output language of the generator (the integrative variabilities). The integrative composition mechanism automatically selects the appropriate alternatives. Since the possible integrations and interferences are known and defined for each language, a developer of an integrative composable generator can anticipate these integrative variabilities.

Domain-specific Integrative Composition The presented integrative composition mechanism is independent of a particular implementation language for generated programs. An integrative composition can be specified in various languages, including domain-specific (modeling) languages. A domain-specific integrative composition offers the opportunity to specify an integration of generated programs at a higher-level of abstraction. In such domain-specific integrative compositions, domain-specific interferences can be detected and resolved whenever possible. A domain-specific integrative composition also ensures that no lower-level composition conflicts exist because the integrated domain-specific program is entirely translated by a single program generator into an executable program. We have also described the use of the Feature Description Language (FDL) as a domain-specific language to express feature configurations. The FDL can be used by all generators in the implementation of a library of generators. It allows to detect and prevent high-level (semantic) conflicts in integrative compositions without the need to define an entire domain-specific language.

Generative Logic Metaprogramming Logic metaprogramming was already identified as a metalanguage that can be used to generate programs [Vol98]. We argue that the logic metaprogramming language is a suitable implementation language for integrative composable program generators. The logic language offers appropriate linguistic support for the implementation of generative programs to handle integrative variabilities. More specifically, the multiple subsequent results that are associated with the invocation of a logic query are a key to the implementation of alternative generative programs. In a single generative logic metaprogram, we can specify multiple alternative declarations that each implement the generation of an alternative generated program part. Each of these alternative declarations can be parameterized with the possible integration and dependence relations imposed on the generated program part. Each declaration can thus implement one or more integrative variabilities. The logic evaluator automatically invokes all alternative declarations until a correct generated program is found. This correctness is checked by a constraint checker, which verifies the dependency and integration relations imposed between all program parts. Generative Logic Metaprogramming is thus a combination of a logic metalanguage and a constraint checking system. This combination proves to be a suitable generative implementation technique to handle integrative variabilities in the implementation of integrative composable generators.

8.3 Future Work

Now that we have presented what we have achieved, it is time to mention what we did not do. We present a number of directions for future research and opportunities for improvements.

8.3.1 Future Research

Unanticipated and Automatic Adaptations

Although the implementation of a generator does not have to implement all possible integrative compositions, the generation of each program part must anticipate the possible integrations for that program part. As a result, an integrative composable generator can only compose if its implementation has anticipated to this and implements the required adaptation. Furthermore, the implementation of many adaptations is identical or similar in many program generators. For example, the implementation that adapts a method program part for the integration in another class is identical in almost any program generator. Although it remains desirable that the implementation of a program generator offers full control over the possible adaptations (and integrations), it makes sense to provide a standard adaptation that can occur automatically. The same is true for adaptations that circumvent the composition conflicts. The renaming of private instance variables, for example, is an adaptation that you would want to occur automatically. Such automatic adaptations can be included in the language definition.

The implementation of automatic adaptations can be even more powerful if the adaptation strategy is not fixed but can be driven by the generators involved in an integrative composition. The adaptation mechanism can then consider information provided by both generators to resolve a composition conflict or to adapt the program for a correct integration. For example, consider the experiment presented in the previous chapter of the domain-specific integrative composition of user interfaces. Each generator provided an adaptation in the specification of the coordinates of the separate UI elements. This allowed that windows merged and that the UI elements did not overlap. However, the adaptations were driven with only 'local' knowledge: i.e. each generator did not know about the desired positions of the UI elements of the other generator. As a result, the adaptations are guesses that try to prevent the composition conflict of overlapping UI elements. If the adaptation mechanism would be provided with more information (of all coordinates), it can find a more optimal solution. However, it requires a language in which each generator can specify its desires, such that the adaptation mechanism can find a solution automatically.

Application to Composable Aspect Weavers

A particular application of composable program generators is the building of composable aspect weavers. Composable aspect weavers for domain-specific aspect languages reconciles the advantages of domain-specific aspect languages with the ability of implementing multiple kinds of aspects in a single application. We can build new aspect weavers for each domain-specific aspect language and we can compose them into a composed weaver that can accept aspects of multiple domain-specific aspect languages. We have already experimented with building composable aspect weavers using LMP in [BMV02]. The technique presented in this dissertation to implement integrative composable generators can also be used to implement composable aspects weavers. However, at the moment, we still need a low-level general purpose aspect language to achieve this. In essence, we can already build composable aspect weavers that translate their program into a single general-purpose aspect language. This is a rather straightforward application of the technique presented in this dissertation onto the experiments presented in [BMV02].

To implement integrative composable generators as 'complete' aspect weavers, we need to extend the concept of integration relations with the concept of crosscuts. Instead of integration relations that can be imposed between two program parts only, we need integration relations that can operate on multiple program parts. We envision that such integration relations are not expressed as single facts in LMP but will rather use logic rules. The logic rule can then express a pointcut. We have also experimented with the use of logic rules to express crosscuts in [GB03]. We envision that both approaches can be combined for the implementation of composable aspect weavers.

Application to Model-driven Architectures

Generative programming techniques are a likely approach for the implementation of model transformations in Model-Driven Architectures (MDA). MDA proposes that platform-independent models (PIMs) are automatically mapped into more platform-specific models (PSMs) and eventually into executable code.

In our opinion, MDA benefits from the implementation of integrative composable generators to implement these mappings. This is because MDA implicitly envisions separate generators (and even separate PIMs) in the mapping from the PIM to the PSM. This is specifically illustrated in the transformation of pervasive services (e.g. transaction, security, etc):

> MDA will provide common, platform independent models of pervasive services. It will provide mappings for transforming models, which draw on these pervasive service PIMs, to platform specific models using the services as provided by a particular platform.

The implementation of these mappings for each pervasive service is best modularized in a separate generator. The generators must, however, produce a common PSM where the pervasive services are integrated. Furthermore, MDA explicitly mentions the combination of mappings into a combined mapping. They distinguish between sequential and concurrent combination, which correspond to our translation and integrative composition.

Application to other Generative Technologies

We have mentioned that the implementation of integrative composable generators is independent of GLMP. In essence, we could extend any generative programming technology with the concepts of integrative composition interfaces, program parts, integration relations, etc.... However, it remains to be researched how feasible and how adequate the result of such an extension would be. This application presents an opportunity for an interesting additional validation of our approach and will most certainly raise new issues that need to be dealt with. Therefore, it remains to be researched how well our approach can integrate in well-established generative programming techniques.

Output Languages

We have consistently used the Smalltalk language as the final output language of all generators in this dissertation. Nevertheless, the GLMP system does not impose the use of the Smalltalk language as the final implementation language of all generated programs. The simplicity of the syntactic structure of Smalltalk was both an advantage and a drawback. On the one hand, Smalltalk programs can be easily represented and not many different program parts are required. On the other hand, the lack of syntactic details in a Smalltalk program severely limits the possible composition conflicts that can be detected and resolved. For example, we have added optional typing information to the representation of variables to detect typing conflicts in the integration of variables.

The application of the GLMP approach to the implementation of program generators with other output languages and especially with output languages in other paradigms (functional, prototype-based, logic, etc) represents an interesting set of future experiments. Moreover, it would be interesting to investigate how the change of final output language would impact the languages used at higher-levels of abstraction. In essence, given a particular domain-specific modeling language, can we replace the generation to Smalltalk with the generation to Java? What is the impact on the possible integrative variabilities?

8.3.2 Improvements

Integrators

An integration specification always consisted out of single relations (implemented with logic facts). Furthermore, an integration specification always occurred in terms of the output programs. Conversely, we did identify useful sets of integration relations that performed particular integrative compositions. For example, for the observer-observable generator, we identified framework composition, aspectual integration, etc.... It would be useful to group sets of integration relations into an *integrator*. An integrator is merely an integration specification that was determined beforehand. Furthermore, we could implement integrators such that they perform a particular integration, based on an input specification. Therefore, the integration relations would be expressed as rules instead of facts. These rules will derive the actual integration specification from the input specification of the integrators. The implementation of such integrators is most useful in a library of program generators where the possible integrative compositions between the program generators are known. For each possible integrative composition, a separate integrator can be written. The integrators and generators can the be used by a developer to compose program generators without requiring knowledge on the integrative composition interfaces. Furthermore, integrators can implement a crosscut language in the context of composable aspect weavers (mentioned earlier).

Integrative Composition Adapters

An integrative composition requires that the generators anticipate possible integrations of their program parts with other program parts. In doing so, a generator always poses restrictions on the possible integrations. For example, a generator can require the integration of a required part with a method part that has a particular method signature. An example of such a composition was illustrated in the integrative composition of the tree generator and the traversal generator (section 5.7.2). This integrative composition required that the traversal generator is able to 'use' the iterator provided by the tree generator. Although the tree generator provided two possible methods for iterating over the child nodes, it still means that the tree generator cannot compose with a traversal generator that expects that a `variable` program part contains the child nodes. This mismatch of the composition interfaces reflects that the generated programs of both generators cannot be integrated. To achieve an integration of the generated programs, some additional glue code needs to be generated. For example, the previous example could be solved by generating an additional variable that can be used by the traversal code. Of course, this variable also needs to contain the child nodes. Therefore, an additional method needs to be generated that populates the variable with the child nodes through the use of the iterator generated by the tree generator. This additional method also needs to be executed *before* the traversal code. Essentially, this requires the inclusion of an additional generator that produces glue-code. Consequently, this solution does not require fundamental changes to our approach. Such a generator can be considered as an *integrative composition adapter*. Its integrative composition interface needs to match the integrative composition interfaces of the generators whose interface mismatch it tries to solve. To a certain extent, the consistency generator (introduced in section 7.5.1) can be seen as a special case of such an integrative composition adapter. The consistency generator produces additional functionality, while an integrative composition adapter would only produce glue-code to allow an integrative composition.

Automatic Application of Dependency and Integration Relations

Our approach requires that each generative program explicitly deals with the dependency and integration relations that are imposed on its generated program part. Therefore, each generative logic metaprogram needs to explicitly call the appropriate predicate associated with the relation. For integration relations this mechanism is required because it is desirable that the generative program itself has control over the program parts to integrate with. However, many integration relations are applied in the same way. For example, all `in` and `subclass` integration relations are handled in the same way in all generative programs implemented in this dissertation. Moreover, in the case of dependency relations, the explicit control of the generative program over the dependency is not required because the dependency is implemented by the generator developer himself. As a result, there is a lot of repetition in the implementation of all generative programs. This could be improved by including standard rules for the application of integration relations and dependency relations in the language definition. The rules are then automatically applied to the program parts. For example, in the case of and `in` dependency or integration relation, the generative program would not explicitly 'call' this relation using the `in/3` predicate to retrieve the class name. Instead, it would leave the class name open in its generated program part, which will be filled in by the standard application rules for the integration relations.

Circular Dependencies Analysis and Resolution

The current implementation of GLMP does not automatically detect circular dependencies in the implementation and composition of program generators. It merely prevents infinite loops through a cut-off depth of the logic evaluator stack. Nevertheless, because all dependencies and integrations are explicitly defined in the implementation and composition of program generators, the system can analyze the dependencies and detect circularities. If a circularity is found, we can decide that the composition fails or we can try to use partial evaluation to resolve the circularity. We have already proposed a 'manual' implementation of such a partial evaluation through the definition of an alternative logic declaration that produces a partial implementation for a program part. This can be used to resolve circularity conflicts manually. However, such a manual approach is not possible (or at least not practically feasible) if circular dependencies arise in an integrative composition. The application of partial evaluation techniques to logic programs could provide a resolution in some cases of circular dependencies.

Composition Language

We do not have a separate composition language that can easily describe the composition of program generators. In the prototype implementation of GLMP, we use Smalltalk as a composition language. The composition of individual program generators and their integration specifications need to be written as Smalltalk expressions. To allow an easy use of the GLMP technique, an appropriate composition language is most desirable.

Beyond a Prototype Implementation

To validate our approach in this dissertation, we have developed a prototype implementation of the GLMP system and a prototype development interface. In essence, a simple constraint checking system was implemented on top of the Soul system together with an appropriate user interface to specify program parts, generative programs, relations and constraints. The use of this GLMP system requires solid knowledge about the Soul language and has no real development support (such as debugging). In the context of the research performed within the European Network of Excellence on AOSD, we will expand our prototype tool. Besides the technical issues in this expansion, we can typically research the application of more advanced constraint-solving techniques.

Appendix A

Language Definitions

A.1 Smalltalk Language

A.1.1 Program Parts

Smalltalk Element	**Logic Representation**
class	class(?name,?superClassName)
variable	var(?className,?name,?Optionaltype)
method	method(?className,?methodName,?methodBody,?info)

A.1.2 Dependency and Integration Relations

Integration Relation	**Logic Declaration**	**ConstraintPredicate**
subclass	subclass(?partA,?partB)	constraintSubclass/4
in	in(?partA,?partB)	constraintVarInClass/4
in	in(?partA,?partB)	constraintMethodInClass/4
overrides	overrides(?partA,?partB)	constraintOverrides/4
unite	unite(?partA,?partB)	constraintUnite/4
includeBefore	includeBefore(?partA,?partB)	constraintIncludeBefore/4
includeAfter	includeAfter(?partA,?partB)	constraintIncludeAfter/4
Dependency Relation	**Logic Declaration**	**ConstraintPredicate**
refers	refers(?partA,?partB)	constraintRefers/4
contains	contains(?partA,?partB)	
self-calls	selfcalls(?partA,?partB)	constraintSelfcalls/4
calls	calls(?partA,?partB)	constraintCalls/4

A.1.3 Dependency Relation Enforcement

```
constraintCalls(?m1,?m2,method(?class1,?method1,?,?info),
                        method(?class2,?method2,?,?))  if
   member(?method2,?info)

constraintIn(?m,?c,method(?class,?,?,?),class(?class,?))
constraintIn(?v,?c,var(?class,?,?),class(?class,?))

constraintOverrides(?m1,?m2,method(?class1,?method,?,?),method(?class2,?method,?,?))

constraintRefers(?m,?v,method(?,?,?,?info),var(?,?var,?type)) if
   member(var(?var,?type),?info)
constraintRefers(?m,?v,method(?,?,?,?info),var(?,?var,?type)) if
   member(?var,?info)
constraintRefers(?m,?c,method(?,?,?,?info),class(?name,?))  if
   member(?name,?info)

constraintSelfcalls(?m1,?m2,method(?class1,?method1,?,?info),
                        method(?class2,?method2,?,?))  if
   member(?method2,?info)

constraintSubclass(?c1,?c2,class(?class1,?class2),class(?class2,?)).

constraintContains(?v,?c,var(?,?,?type),class(?type,?)).
```

A.1.4 Integration Relation Enforcement

```
constraintUnite(?p1,?p2,?val,?val)

constraintIncludeAfter(?m1,?m2,method(?class,?method1,?body1,?info1),
                        method(?class,?method2,?body2,?info2)) if
   or(unarySelector(?method2),sameSignature(?method1,?method2)),
   includedAfter(?body1,?body2),
   foreach(member(?x,?info2),member(?x,?info1))
```

A.1.5 Composition Conflict Detection

```
classConstraint(?nameA,?nameB,class(?classA,?superA),class(?classB,?superB)) if
   not(or(unite(?nameA,?nameB),unite(?nameB,?nameA))),
   not(equals(?classA,?classB))
classConstraint(?nameA,?nameB,class(?classA,?superA),class(?classB,?superB)) if
```

```
   or(unite(?nameA,?nameB),unite(?nameB,?nameA))

methodOverridesConstraint(?nameA,?nameB,method(?classA,?methodA,?bodyA,?infoA),
                             method(?classB,?methodB,?bodyB,?infoB)) if
   not(overrides(?nameA,?nameB)),
   not(and(inSameHierarchy(?classA,?classB),equals(?methodA,?methodB)))
methodOverridesConstraint(?nameA,?nameB,method(?classA,?method,?bodyA,?infoA),
                             method(?classB,?method,?bodyB,?infoB)) if
   or(overrides(?nameA,?nameB),overrides(?nameB,?nameA))

varShadowsConstraint(?nameA,?nameB,var(?classA,?varA,?),var(?classB,?varB,?))  if
   not(and(inSameHierarchy(?classA,?classB),equals(?varA,?varB)))

methodUnitesConstraint(?nameA,?nameB,method(?classA,?methodA,?bodyA,?infoA),
                             method(?classB,?methodB,?bodyB,?infoB)) if
   not(or(unite(?nameA,?nameB),unite(?nameB,?nameA))),
   not(and(equals(?classA,?classB),equals(?methodA,?methodB)))
methodUnitesConstraint(?nameA,?nameB,method(?classA,?methodA,?,?),
                             method(?classB,?methodB,?,?))  if
   or(unite(?nameA,?nameB),unite(?nameB,?nameA))

varUnitesConstraint(?nameA,?nameB,var(?classA,?varA,?typeA),var(?classB,?varB,?typeB)) if
   not(or(unite(?nameA,?nameB),unite(?nameB,?nameA))),
   not(and(equals(?classA,?classB),equals(?varA,?varB)))
varUnitesConstraint(?nameA,?nameB,?,?)  if
   or(unite(?nameA,?nameB),unite(?nameB,?nameA))
```

A.1.6 Integration Relation Propagation

```
sub(?c,?d) if
   overrides(?a,?b),
   ?genA->in(?a,?c),
   ?genB->in(?b,?d)

unite(?c,?b) if
   sub(?a,?b),
   ?genA->sub(?a,?c)

unite(?c,?b) if
   in(?a,?b),
   ?genA->in(?a,?c)

unite(?c,?d) if
```

```
   unite(+?a,+?b),
   ?genA->in(?a,?c),
   ?genB->in(?b,?d)

unite(?c,?d) if
   unite(+?a,+?b),
   ?genA->sub(?a,?c),
   ?genB->sub(?b,?d)
```

A.1.7 Additional Metaprograms

Some of the additional metaprograms use the symbiosis of Soul with Smalltalk. They consequently are partially implemented in Smalltalk.

```
doubleKeyword(?x) if
   equals([?x occurrencesOf:  $:],2)

makeMethodHeader(?selector,?args,?header) if
   equals(?selColl,[newSelector := WriteStream on:  String new.
                   selectorColl := OrderedCollection new.
                   ?selector asString do:[:char | newSelector nextPut:  char.
                        char = $:  ifTrue:[ selectorColl add:  (newSelector contents).
                        newSelector := WriteStream on:  String new]].
                   selectorColl]),
   listAsCollection(?args,?argsColl),
   equals(?header,[ header := String new.
              ?selColl with:?argsColl do:[:sel:arg|header:=header,sel,' ',arg,' '].header]).

sameSignature(?messageA,?messageB) if
   equals([?messageA occurrencesOf:  $:],[?messageB occurrencesOf:  $:])

singleKeyword(?x) if
   equals([?x occurrencesOf:  $:],1)

unaryMessage(?x) if
   equals([?x occurrencesOf:  $:],0)

inSameHierarchy(?classNameA,?classNameB) if
   generated(class,class(?classNameA,?superA)),
   generated(class,class(?classNameB,?superB)),
   or(inSameHierarchySub(class(?classNameA,?superA),
         class(?classNameB,?superB)),
   inSameHierarchySub(class(?classNameB,?superB),class(?classNameA,?superA)))
inSameHierarchySub(class(?classA,?classC),?classB) if
   generated(class,class(?classC,?classD)),
   inSameHierarchySub(class(?classC,?classD),?classB)
inSameHierarchySub(class(?classA,?classB),class(?classB,?))

includedAfter(?body,?includeBody) if
   argumentList(?body,?arguments),
   convertAndStripHeader(?includeBody,?arguments,?convertedBody),
   [(ParseTreeSearcher treeMatching:  '  '@.Statements.  ?convertedBody' in:
                   (RBParser parseMethod:  ?body))    nil ]
```

A.2 Feature Description Language

A.2.1 Program Parts

FDL Element	**Logic Representation**
featural description	features(?name,?featureNameList)

A.2.2 Dependency and Integration Relations

Integration Relation	**Logic Declaration**	**ConstraintPredicate**
compatible(?x)	compatible(?partA,?partB,?x)	constraintCompatible/4
unite	unite(?partA,?partB)	constraintUnite/4
Dependency Relation	**Logic Declaration**	**ConstraintPredicate**
requires(?x,?y)	requires(?partA,?partB,?x,?y)	constraintRequires/4
excludes(?x,?y)	excludes(?partA,?partB)	constraintExcludes/4

A.2.3 Dependency Relation Enforcement

```
constraintExcludes(?nameA,?nameB,features(?,?featuresA),features(?,?featuresB)) if
   forall(and(excludes(?nameA,?nameB,?fA,?fB),
                   member(?fA,?featuresA)),not(member(?fB,?featuresB))).

constraintRequires(?nameA,?nameB,features(?,?featuresA),features(?,?featuresB)) if
   forall(and(requires(?nameA,?nameB,?fA,?fB),member(?fA,?featuresA)),
              member(?fB,?featuresB))
```

A.2.4 Integration Relation Enforcement

```
constraintUnite(?nameA,?nameB,features(?,?featuresA),features(?,?featuresB)) if
   sameElements(?featuresA,?featuresB).
```

A.2.5 Additional Metaprograms

```
valid(features(?,?features)) if
   completeFeatures(?features),
   forall(and(member(?f,?features),excludes(?f,?g)),not(member(?g,?features))),
   forall(and(member(?f,?features),requires(?f,?g)),member(?g,?features)).

completeFeatures(?feature,?l) if
   oneOf(?feature,?list),
   member(?f,?list),
   completeFeatures(?f,?l).

completeFeatures(?feature,?completeList) if
   all(?feature,?list),
   findall(?solutions,and(member(?f,?list),
                  findall(?sol,completeFeatures(?f,?sol),?solutions)),
         ?allSolutions),
   map(?aSolution,?allSolutions,member),
   flatten(?aSolution,?completeList).

completeFeatures(?f,<?f>) if
   not(or(oneOf(?f,?),moreOf(?f,?),all(?f,?))).

completeFeatures(?feature,?completeList) if
   all(?feature,?list),
   findall(?solutions,
         and(member(?f,?list),not(optional(?f)),findall(?s,completeFeatures(?f,?s),?solutions)),
         ?allSolutions),
   map(?aSolution,?allSolutions,member),
   flatten(?aSolution,?completeList).

completeFeatures(?feature,?completeList) if
   moreOf(?feature,?list),
   sublistOf(?sublist,?list),
   not(equals(?sublist,<>)),
   findall(?solutions,
         and(member(?f,?sublist),findall(?sol,completeFeatures(?f,?sol),?solutions)),
         ?allSolutions),
   map(?aSolution,?allSolutions,member),
  flatten(?aSolution,?completeList)
```

Appendix B

Observer-observable Generator

B.1 Generative Programs

B.1.1 AddObserver Part

```
addObserver(method(?class,{addObserver:},{addObserver:  anObserver
                                  ?observers add:anObserver},<var(?observers,?type)>)) if
   in(addObserver,Observable,class(?class,?)),
   refers(addObserver,observers,var(?,?observers,?type)),
   observersType(?type).

addObserver(method(?class,{addObserver:},{addObserver:  anObserver
                                       (?observers includes:  anObserver)
                                       ifFalse:[?observer add:  anObserver]},
                                       <var(?observers,?type)>)) if
   in(addObserver,Observable,class(?class,?)),
   refers(addObserver,observers,var(?,?observers,OrderedCollection)),
   observersType(Set)
```

B.1.2 Changed Part

```
changed(method(?class,{changed:},{changed:anAspect
                                       ?observers do:[:observer | observer ?updateMessage]},
                                 <?observers,?updateSelector>)) if
   in(changed,?,class(?class,?)),
   refers(changed,observers,var(?class,?observers,?)),
   calls(changed,update,method(?,?updateSelector,?,?)),
   updateMessage(?updateSelector,?updateMessage).
```

B.1.3 Initializer Part

```
initializer(method(?class,initialize,{initialize ?observers:= ?type new},
                                <var(?observers,?type)>)) if
   observersType(?type),
   in(initializer,Observable,class(?class,?)),
   refers(initializer,observers,var(?class,?observers,?type)).
```

B.1.4 Observable Part

```
Observable(class(Observable,Object)).

Observable(class(?name,?super)) if
   unite(Observable,?,class(?name,?super)).

Observable(class(?name,?super)) if
   sub(Observable,?,class(?super,?))
```

B.1.5 ObservableVar Part

```
observableVar(<var(?class,observable,nil)>) if
   feature(instancevariable),
   in(observableVar,Observer,class(?class,?)).

observable(<?var>) if
   unite(<?var>),
   not(feature(instancevariable)).

observableVar(<>) if
   not(feature(instancevariable)).
```

B.1.6 Observer Part

```
Observer(class(Observer,Object)).

Observer(class(?name,?super)) if
   unite(Observer,?,class(?name,?super)).

Observer(class(Observer,?super)) if
   sub(Observer,?,class(?super,?)).
```

B.1.7 Observers Part

```
observers(var(?class,?name,?type)) if
   in(observers,Observable,class(?class,?)),
   observersName(?name),
   observersType(?type).

observers(var(?class,?name,?type)) if
   in(observers,Observable,class(?class,?)),
   unite(observers,?,var(?class,?name,?type)).
```

B.1.8 Update Part

```
update(method(?class,{update:},{update:  anAspect },<>)) if
   not(feature(parameter)),
   in(update,?,class(?class,?)).

update(method(?class,?selector,?body,?info)) if
   not(feature(parameter)),
   in(update,?,class(?class,?)),
   unite(update,?,method(?,?selector,?body,?info)),
   singleKeyword(?selector).

update(method(?class,{update:from:},{update:anAspect from:anObservable
                                     ^ self subclassResponsability},<>)) if
   feature(parameter),
   in(update,?,class(?class,?)).

update(method(?class,?selector,?body,?i)) if
   feature(parameter),in(update,?,class(?class,?)),
   unite(update,?,method(?,?selector,?body,?i)),
   doubleKeyword(?selector).
```

B.1.9 Additional Programs

```
observersName(observers).
observersName(observersList).
observersType(Set) if feature(single).
observersType(OrderedCollection) if feature(multi).
outputLanguage([GenLogica.SmalltalkLanguage]).
updateMessage(?updateSelector,{?updateSelector anAspect}) if
```

```
   not(feature(parameter)).
updateMessage(?updateSelector,?updateMessage) if
   feature(parameter),
   makeMethodHeader(?updateSelector,<anAspect,self>,?updateMessage)
```

B.2 Dependency Relations

```
calls(changed,update).
in(initializer,Observable).
in(changed,Observable).
in(update,Observer).
in(observableVar,Observer).
in(addObserver,Observable).
in(delObserver,Observable).
in(observers,Observable).
refers(addObserver,observers).
refers(delObserver,observers).
refers(initializer,observers).
refers(changed,observers).
```

B.3 Integrative Compositions

B.3.1 Framework Specialization

```
overrides(updateB,update).
overrides(updateA,update).
sub(ObserverA,Observer).
sub(ObserverB,Observer).
sub(MyModel,Observable).
unite(publish,changed).
unite(model,observableVar).
```

B.3.2 Aspectual Integration

```
unite(publish,changed).
unite(model,observable).
unite(update,updateA).
unite(Observer,ObserverA).
unite(Observable,MyModel).
```

Appendix C

Data Container and Consistency Generators

C.1 Data Container Generator

C.1.1 Generative Programs

Initialize Method Part

```
initialize(method(?class,?selector,?body,<?contents,?index>)) if
   in(initialize,DataContainer,class(?class,?)),
   refers(initialize,Index,var(?,?index,?)),
   refers(initialize,Contents,var(?,?contents,?)),
   initializeMethod(?contentsVar,?indexVar,?selector,?args,?body)
```

Add Method Part

```
Add(method(?class,?selector,{?header ?temporaries ?includeBefore ?body ?includeAfter },
         <?contents,?index>)) if
   in(Add,DataContainer,class(?class,?)),
   refers(Add,Contents,var(?,?contents,?)),
   refers(Add,Index,var(?,?index,?)),
   addMethod(?contents,?index,?selector,?args,?body,?tempsList),
   includeBefore(Add,?selector,?args,?includeBefore,?includeBeforeInfo),
   includeAfter(Add,?selector,?args,?includeAfter,?includeAfterInfo),
   makeMethodHeader(?selector,?args,?header),
   makeTemporariesList(?tempsList,?temporaries)
```

Remove Method Part

```
Remove(method(?class,{remove:},{remove:anEl ?temporaries ?includeBefore ?body ?includeAfter },
         <?contents,?index>)) if
   in(Remove,DataContainer,class(?class,?)),
   refers(Remove,Contents,var(?,?contents,?)),
   refers(Remove,Index,var(?,?index,?)),
   removeMethod(?contents,?index,anEl,?body,?tempsList),
   includeBefore(Remove,{remove:},<anEl>,?includeBefore,?includeBeforeInfo),
   includeAfter(Remove,{remove:},<anEl>,args,?includeAfter,?includeAfterInfo),
   makeTemporariesList(?tempsList,?temporaries)
```

Iterator Method Part

```
Iterator(method(?class,{do:},{do:aBlock 1 to:?indexVar-1 do:[:index|aBlock
         value:((?contents at:2) at:index] },<?contents,?indexVar>)) if
   in(Iterator,DataContainer,class(?class,?)),
   refers(Iterator,Contents,var(?,?contents,?)),
   refers(Iterator,IndexVar,var(?,?indexVar,?)),
   feature(fixed).
Iterator(method(?class,{do:},{do:aBlock |current| current:=?contents.[(current key=nil)]
         whileFalse:[aBlock value:current key.  current:=current value].},
         <?contents,?indexVar>)) if
   in(Iterator,DataContainer,class(?class,?)),
   refers(Iterator,Contents,var(?,?contents,?)),
   refers(Iterator,IndexVar,var(?,?indexVar,?)),
   feature(plain)
```

Contents Variable Part

```
Contents(var(?class,contents,<>)) if
   in(Contents,DataContainer,class(?class,?)).
Contents(var(?class,collection,<>)) if
   in(Contents,DataContainer,class(?class,?))
```

Index Variable Part

```
Index(var(?class,index,<>)) if
   in(Index,DataContainer,class(?class,?)).
Index(var(?class,theIndex,<>)) if
   in(Index,DataContainer,class(?class,?))
```

DataContainer Class Part

```
DataContainer(class(?name,Object)) if
   containerName(?name).
DataContainer(class(?name,?superclass)) if
   sub(DataContainer,?,class(?superclass,?)),
   containerName(?name).
DataContainer(?class) if
   unite(DataContainer,?,?class)
```

Additional Logic Programs

```
addBodyKeyValue(?contentsVar,?indexVar,?keyVar,?elVar,{
         (?contentsVar at:  1) doWithIndex:[:key:index|key=?keyVar ifTrue:[setIndex:=index]].
         setIndex = nil ifTrue:[(?indexVar > (?contentsVar at:  1) size)
         ifTrue:[self error:'overflow'].
         setIndex := ?indexVar.  ?indexVar := ?indexVar + 1].
         (?contentsVar at:  1) at:  setIndex put:  ?keyVar.
         (?contentsVar at:  2) at:  setIndex put:  ?elVarExpression},<setIndex>) if
   feature(fixed),
   addElement(?elVar,?elVarExpression).
addBodyKeyValue(?contentsVar,?indexVar,?keyVar,?elVar,{
         current := ?contentsVar.
         [(current key = nil)  (current key = ?keyVar)]
         whileFalse:[current := current value].
         current key key:  ?keyVar.
         current key value:  ?elVarExpression.
         current value = nil ifTrue:[current value:  (nil->nil)]},<current>) if
   feature(growable),
   addElement(?elVar,?elVarExpression).

addBodyPlain(?contentsVar,?indexVar,?elVar,{ (?indexVar > ?contentsVar size)
         ifTrue:[self error:'overflow'].
         ?contentsVar at:  ?indexVar put:  ?elVarExpression.
         ?indexVar := ?indexVar + 1.},<>) if
   feature(fixed),
   addElement(?elVar,?elVarExpression).
addBodyPlain(?contentsVar,?indexVar,?elVar,{ current := ?contentsVar.
         currentIndex := 1.
         [currentIndex = ?posVar]
         whileFalse:[ currentIndex := currentIndex + 1.
         (current value = nil)
         ifTrue:  [current value:  (nil->nil)].
         current := current value].
         current key:  ?elVarExpression.
         ?indexVar := ?indexVar + 1},<current,currentIndex>) if
   feature(growable),
   addElement(?elVar,?elVarExpression).

addElement(?elVar,{?elVar copy}) if
   feature(copy).
addElement(?elVar,{?elVar}) if
   feature(reference).

addMethod(?contentsVar,?indexVar,{add:},<anEl>,?addBody,?temps) if
   feature(plain),
   addBodyPlain(?contentsVar,?indexVar,anEl,?addBody,?temps).
addMethod(?contentsVar,?indexVar,{at:put:},<aKey,anEl>,?addBody,?temps) if
   feature(keyvalue),
   addBodyKeyValue(?contentsVar,?indexVar,aKey,anEl,?addBody,?temps).

containerName(DataContainer).
containerName(Container).

initializeMethod(?contentsVar,?indexVar,{initialize:},<aMax>,{initialize:  aMax
                   ?contentsVar := Array new:  aMax.  ?indexVar := 1.}) if
   feature(fixed),
   feature(plain).
initializeMethod(?contentsVar,?indexVar,{initialize},<>,
                   {initialize ?contentsVar := (nil -> nil).  ?indexVar := 0.}) if
   feature(growable).
initializeMethod(?contentsVar,?indexVar,{initialize:},<aMax>,{initialize:  aMax
         ?contentsVar := Array new:  2.
         ?contentsVar at:  1 put:  (Array new:  aMax).
         ?contentsVar at:  2 put:  (Array new:  aMax).
         ?indexVar := 1.}) if
   feature(fixed),
   feature(keyvalue).

removeBody(?contentsVar,?elVar,{?contentsVar remove:  ?elVar}) if
   feature(plain).
removeBody(?contentsVar,?keyVar,{?contentsVar removeKey:  ?keyVar}) if
   feature(keyvalue).

sizeExpression(?contentsVar,?indexVar,{?contentsVar size}) if
   feature(fixed),
   feature(plain).
sizeExpression(?contentsVar,?indexVar,{(?contentsVar at:  1) size}) if
```

```
   feature(fixed),
   feature(keyvalue).
sizeExpression(?contentsVar,?indexVar,{?indexVar}) if
   feature(growable)
```

C.2 Consistency Generator

C.2.1 Generative Programs

C.2.2 Required Parts

All required parts are implemented with a similar rule. We show the `ContainA` required part as an example.

```
ContainA(?class) if
   unite(ContainA,?,?class)
```

C.2.3 LockA variable part

The implementation of `LockB` is identical.

```
lockA(var(?class,sema,<>)) if
   in(lockA,ContainA,class(?class,?)).

lockA(var(?class,semaphore,<>)) if
   in(lockA,ContainA,class(?class,?)).

lockA(var(?class,semaVariable,<>)) if
   in(lockA,ContainA,class(?class,?))
```

C.2.4 SyncVarA variable part

The implementation of `SyncVarB` is identical.

```
syncVarA(var(?class,datacontainers,<>)) if
   in(syncVarA,ContainA,class(?class,?)).

syncVarA(var(?class,containers,<>)) if
   in(syncVarA,ContainA,class(?class,?)).
```

```
syncVarA(var(?class,syncContainers,<>)) if
   in(syncVarA,ContainA,class(?class,?))
```

C.2.5 SendAddA method part

The implementation of `SendAddB` is identical.

```
SendAddA(method(?class,?selector,{?header sema ifFalse:[ ?sema := true.
                   (?syncVar at:  ?class) do:  [:c  c ?sendContainAMessage].
                   (?syncVar at:  ?containB) do:  [:c  c ?sendContainBMessage].
                   ?sema := false] },<?selectorA,?selectorB,?sema,?syncVar,?containB>)) if
   in(SendAddA,ContainA,class(?class,?)),
   refers(SendAddA,ContainB,class(?containB,?)),
   refers(SendAddA,syncVarA,var(?,?syncVar,?)),
   refers(SendAddA,lockA,var(?,?sema,?)),
   calls(SendAddA,addA,method(?,?selectorA,?,?)),
   calls(SendAddA,addB,method(?,?selectorB,?,?)),
   sendAddHeader(featA,?selector,?args,?header),
   sendAdd(featA,featA,?args,?sendContainAMessage),
   sendAdd(featA,featB,?args,?sendContainBMessage).
```

C.2.6 SendRemoveA method part

The implementation of `SendRemoveB` is identical.

```
SendRemoveA(method(?class,{sendRemove:},{sendRemove:  anEl
                   ?sema ifFalse:[?sema := true.
                   (?syncVar at:  ?class) do:  [:c  c ?selectorA anEl].
                   (?syncVar at:  ?containB) do:  [:c  c ?selectorB anEl].
                   ?sema := false] },<?selectorA,?selectorB,?sema,?syncVar,?containB>)) if
   in(SendRemoveA,ContainA,class(?class,?)),
   refers(SendRemoveA,ContainB,class(?containB,?)),
   refers(SendRemoveA,syncVarA,var(?,?syncVar,?)),
   refers(SendRemoveA,lockA,var(?,?sema,?)),
   calls(SendRemoveA,removeA,method(?,?selectorA,?,?)),
   calls(SendRemoveA,removeB,method(?,?selectorB,?,?)).
```

C.2.7 SyncA method part

The implementation of `SyncB` is identical.

```
SyncA(method(?class,{syncWith:},{syncWith:  aCont
                   ?sync = nil ifTrue:[?sync := Dictionary new.
                   ?sync at:  ?class put:  Collection new.
                   ?sync at:  ?containB put:  Collection new.]
                   (?sync at:  aCont class) add:  aCont},<?sync,?containB>)) if
   in(SyncA,?,class(?class,?)),
   refers(SyncA,ContainB,class(?containB,?)),
   refers(SyncA,SyncVarA,var(?,?sync,?)).
```

C.2.8 Additional Programs

```
sendAddHeader(?feat,{sendAdd:},<anEl>,?header) if
   feature(?feat,plain).
sendAddHeader(?feat,{sendAt:put:},<aKey,anEl>,?header) if
   feature(?feat,keyvalue).
sendAdd(?featFrom,?featTo,?selectorTo,<?anElVar>,?message) if
   feature(?featFrom,plain),
   feature(?featTo,keyvalue),
   makeMethodHeader(?selectorTo,<{(?anElVar asKey)},?anElVar>,?message).
sendAdd(?featFrom,?featTo,?selectorTo,?args,?message) if
   feature(?featFrom,plain),
   feature(?featTo,plain),
   makeMethodHeader(?selectorTo,?args,?message).
sendAdd(?featFrom,?featTo,?args,?selectorTo,?message) if
   feature(?featFrom,keyvalue),
   feature(?featTo,keyvalue),
   makeMethodHeader(?selectorTo,?args,?message).
sendAdd(?featFrom,?featTo,<?,?anElVar>,?selectorTo,?message) if
   feature(?featFrom,keyvalue),
   feature(?featTo,plain),
   makeMethodHeader(?selectorTo,<?anElVar>,?message)
```

C.3 Dependency Relations

```
calls(SendAddA,addB).
calls(SendAddA,addA).
calls(SendRemoveA,removeA).
calls(SendRemoveA,removeB).
calls(SendRemoveB,removeA).
calls(SendRemoveB,removeB).
in(SyncA,ContainA).
in(SyncB,ContainB).
in(SendAddA,ContainA).
in(SendAddB,ContainB).
in(SendRemoveA,ContainA).
in(SendRemoveB,ContainB).
refers(SendAddA,ContainB).
refers(SendAddA,lockA).
refers(SendAddB,lockB).
refers(SendAddB,addA).
refers(SendAddB,addB).
refers(SyncA,ContainB).
refers(SyncA,syncVarA).
refers(SyncB,ContainA).
refers(SyncB,syncVarB).
refers(SendAddA,syncVarA).
refers(SendAddB,syncVarB).
refers(SendRemoveA,lockA).
refers(SendRemoveA,syncVarA).
refers(SendRemoveB,ContainA).
refers(SendRemoveB,lockB).
refers(SendRemoveB,syncVarB)
```

Bibliography

[ADK$^+$98] William Aitken, Brian Dickens, Paul Kwiatkowski, Oege de Moor, David Richter, and Charles Simonyi. Transformation in Intentional Programming. In *Proceedings of the 5th Int. Conf. on SoftwareReuse*. IEEE, 1998.

[Ass04] Uwe Assman. *Invasive Software Composition*. Springer-Verlag, 2004.

[ASU86] Aho, Sethi, and Ullman. *Compilers: Principles Techniques and Tools*. Addison-Wesley, 1986.

[Bes99] C. Bessiere. Non-Binary Constraints. In *Proceedings of Principles and Practice of Constraint Programming*, 1999.

[BG97] Don Batory and Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, 1997.

[BGW02] Johan Brichau, Kris Gybels, and Roel Wuyts. Towards linguistic symbiosis of an object-oriented and a logic programming language. In *Proceedings of Workshop on Multi-paradigm Programming in OO at ECOOP 2002*, 2002.

[Big94] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In W. B. Frakes, editor, *3rd International Conference on Software Reusability*, pages 102–109. IEEE Press, 1994.

[Big98] T.J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5:169–226, 1998.

[Big00] T. J. Biggerstaff. Reuse technologies and niches, June 2000.

[BJMvH02] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–214, April 2002.

[BLHM02] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *Automated Software Engineering Conference*, pages 81–92, 2002.

[BMV02] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages using logic metaprogramming. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of GPCE Conference*, volume 2487 of *LNCS*, pages 110–127. Springer-Verlag, 2002.

[Boo87] Grady Booch. *Software Components with Ada*. Benjamin/Cummings, 1987.

[BR] John Brant and Don Roberts. Smalltalk compiler compiler. http://www.refactory.com/Software/SmaCC/.

[BSR03] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling Step-wise Refinement. In *International Conference on Software Engineering (ICSE)*, 2003.

[BSST93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. *ACM SIGSOFT 93: Symposium on the Foundations of Software Engineering*, December 1993.

[BST$^+$94a] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. Achieving reuse with software system generators. *IEEE Software*, September 1994.

[BST$^+$94b] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generators. *IEEE Softw.*, 11(5):89–94, 1994.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *2003 Workshop on Generative Techniques in the Context of Model-Driven Architectures*, 2003.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the OOPSLA Conference*, pages 285–299, 1995.

[Cle88] J. Craig Cleaveland. Building application generators. *IEEE Software*, July 1988.

[Cza] Krzysztof Czarnecki. GCSE working group. http://www-ia.tu-ilmenau.de/ czarn/generate/engl.html.

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-based Component Models.* PhD thesis, Technical University of Ilmenau, 1998.

[DD02] Maja D'Hondt and Theo D'Hondt. The tyranny of the dominant model decomposition. In *Workshop on Generative Techniques in the Context of MDA*, 2002.

[DGJ04] Maja D'Hondt, Kris Gybels, and Viviane Jonckers. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004), Special track on Object-Oriented Programming, Languages and Systems.* ACM, March 2004.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DVMW00] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-evolution of Object-Oriented Software Design and Implementation. In *Proceedings of the International Symposium on Software Architectures and Component Technology 2000*, 2000.

[ea90] Kang et al. Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[Fil02] Robert E. Filman. What is Aspect-oriented Programming, Revisited. In *Workshop on Advanced Separation of Concerns, ECOOP 2001, Budapest*, 2002.

[Fla94] Peter Flach. *Simply Logical: Intelligent Reasoning by Example.* Wiley & Sons, 1994.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999. FOW m 01:1 1.Ex.

[FS00] Martin Fowler and Kendall Scott. *UML Distilled.* ot. Addison-Wesley, 2 edition, 2000.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of AOSD 2003.* ACM Press, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable Object-Oriented software.* Addison-Wesley, 1995.

[GR83] Adele Goldberg and Dave Robson. *Smalltalk-80: the language.* Addison-Wesley, 1983.

[Gro] Object Management Group. Model-driven architecture. http://www.omg.org/mda/.

[Has] Haskell. http://www.haskell.org/.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming - a critique of pure objects. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications.* ACM, 1993.

[JCC] Java Compiler Compiler - The Java Parser Generator. http://javacc.dev.java.net/.

[JF88] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1988.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised(5) report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, June 1997.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[LMB] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly.

[Lop97] Cristina Isabel Videira Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, Northeastern University, nov 1997.

[Mic] Microsoft. Using the codedom, in the .net framework developer's guide.

[MKL97] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case study for Aspect-oriented Programming. Technical Report SPL97-009 P9710044, Xerox Palo Alto Research Center, 1997.

[MMW02] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting Software Development through Declaratively Codified Programming Patterns. *Elsevier Journal on Expert Systems with Applications*, pages 405–431, November 2002.

[MPG03] Kim Mens, Bernard Poll, and Sebastian Gonzales. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 169–178. IEEE Computer Society, 2003.

[MSvW01] Oege De Moor, Ganesh Sittampalam, and Eric van Wyk. Intentional programming: a host of language features. Technical report, Oxford University Computing Laboratory, 2001.

[Nei80] James Neighbors. *Software Construction using Components.* PhD thesis, University of California, Irvine, 1980.

[Nei89] J. M. Neighbors. Draco: a method for engineering reusable software systems. *Software reusability: vol. 1, concepts and models*, pages 295–319, 1989.

[New] Jeff Newbern. All about Monads. http://www.nomaware.com/monads/html/.

[OKK$^+$96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3), 1996.

[OT99] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical report, IBM T.J. Watson research center, apr 1999.

[Pre97] Christian Prehofer. Feature-oriented Programming: A Fresh Look at Objects. *Lecture Notes in Computer Science*, 1241, 1997.

[Riv96] Fred Rivard. Smalltalk: A Reflective Language. In *Reflection'96*, 1996.

[SB97] Yannis Smaragdakis and Don Batory. DiSTiL: A Transformation Library for Data Structures. In *Domain-Specific Languages (DSL) Conference*, pages 257–270, 1997.

[SB98] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[SB00] Yannis Smaragdakis and Don Batory. Application Generators. *Encyclopedia of Electrical and Electronics Engineering*, 2000.

[SJ95] Yellamraju V. Srinivas and Richard Jullig. Specware: Formal Support for Composing Software. In *Mathematics of Program Construction*, pages 399–422, 1995.

[Smi90] Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[Sou] The Smalltalk Open Unification Language (SOUL). http://prog.vub.ac.be/research/DMP/soul/soul2.html.

[Ste94] Guy L. Steele, Jr. Building Interpreters by Composing Monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press.

[SZD00] V. Stuikys, G. Ziberkas, and R. Damasevicius. The Language-centric Program Generator Models: 3L Paradigm. *INFORMATICA*, 11(3):325–348, 2000.

[Szy98] C. Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, 1998.

[TBKG04] Tom Tourwe, Johan Brichau, Andy Kellens, and Kris Gybels. Induced Intentional Software Views. *Special Edition of Elsevier's Computer Languages, Systems & Structures Journal*, 30(1-2), 2004.

[TCKI00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A Class-based Macro System for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[Tid01] Doug Tidwell. *XSLT*. O'Reilly, 2001.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[Unk] Unknown. Program Transformation Wiki. http://www.program-transformation.org.

[VD98] Kris De Volder and Theo D'Hondt. Aspect-oriented Logic Metaprogramming. In *Proceedings of Reflection 1998*, 1998.

[vDK02] A. van Deursen and P. Klint. Domain-specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[vKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific Languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[Voe] Markus Voelter. Program Generation: a Survey of Techniques and Tools. http://www.voelter.de/conferences/tutorials.html.

[Vol] Kris De Volder. TyRuBa. http://tyruba.sourceforge.net/.

[Vol98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[vWV03] Jonne van Wijngaarden and Eelco Visser. Program Transformation Mechanics. A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, 2003.

[Wad90] P. L. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[Wuy98] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *Proceedings of TOOLS USA*, pages 112–124, 1998.

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