Milestone1: Software Automation
Survey of Techniques
and
Evolution & Composability Issues

Johan Brichau
johan.brichau@vub.ac.be

Thomas Cleenewerck
thomas.cleenewerck@vub.ac.be

Dirk Deridder
dirk.deridder@vub.ac.be

SAKE - Software Automation & Knowledge Engineering
Advanced Media Project

Contents

Chapter 1

Introduction

Computer software is built to automate labour-intensive, repetitive and complex tasks. Automatisation of these tasks quite 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. The classic 'waterfall' model for software engineering relies on the fact that a domain analyst summarizes the specifications of the task (specified by domain experts) and communicates them to the software developers. The software is then commonly implemented using general-purpose programming languages and through reuse of existing software artifacts available in libraries. However, this process consumes quite some time as the domain analyst and the development team have to understand the concepts of the domain expert very well and also need to have a solid technical background in the chosen implementation technology. Furthermore, repetitive tasks are also very common in the development of software itself. Developers often need to write identical or similar pieces of code in different software applications. This also provides those developers with ample opportunities to make the same mistakes over and over again. Last but not least, in most cases, the initial version of the software rarely meets the expert's expectations.

Program generation and *Domain-specific Languages* are related technologies that bridge the gap between domain experts and developers through implementation automatisation of similar and frequently needed programs. The generation of program parts or the generation of entire software applications introduces automatisation in the software development process itself. A program generator, or an *automated programmer*, is a software program that is implemented once and can be applied to generate the same or similar programs many times with the same reliability. In most cases, the program generator is a compiler for a domain-specific language. In a domain-specific language, language constructs are offered to the developer that are more suited to express a program in the required problem domain. Thus, a program generator will produce an executable program for a specification or a domain-specific program supplied by the developer. This brings program development closer to the domain expert and may even allow him to specify an entire program.

In this survey, we first classify and summarize engineering methods to build program generators, followed by a description of some of their shortcomings with respect to evolution and composition. Program generation is, of course, quite a broad concept in computer science. The compilation of a program written in any programming language to bytecode is quite often referred to as program generation or code generation. On the other end of the spectrum, the automatic derivation of algorithms from declarative, semantic specifications is also called program generation or program synthesis [26, 27]. Therefore, we first define the kind of program generation that is considered in this survey.

1.1 Definition

Program generation 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. 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. The advent of generative programming [10], product-line architectures [3, 4], MDA [14] and knowledge-based software engineering 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....

In essence, any program that produces program code as output can be called a program generator. Of course, most program generators produce a program based on some input specification, commonly supplied by the developer. Moreover, the input specification is often a program itself, meaning that such program generators are actually program transformers: they transform an input program into an output program. Program transformation is defined as: *The systematic development of efficient programs from high-level specifications by meaning-preserving program manipulations. Also known as optimisation* [12]. In fact, many program generators are program transformers but in program generators, the difference between the input language is very different from the output language. As such, program transformations such as refactorings cannot be considered as program generation. Program generators are also often seen as compilers for domain-specific languages. In many cases, the input specifications are written in a high-level language specifically designed to express abstractions applicable to 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 survey, we focus on program generators in the context of generative programming, product-line architectures and implementations for domain-specific languages (DSLs)[31]. Czarnecki [10] defined generative programming as follows:

> Generative programming is a software-engineering paradigm based on modeling software families such that, given a particular requirements specification, a higly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

This definition implicitly assumes that the generated *end-product* is a completely functional software component. This is quite important as it means that we consider that a generated part implements its complete functionality such that the developer is not required to modify the generated part once 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 reuse techniques, 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* mostly is the input specification for the program generator and is specified by the developer. 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 [9]:

> 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 text, program generators are software programs that produce a finished end-product that cannot and should not be changed by hand to render the end-product usable in a particular application. This is an obvious requirement when using program generators as implementations for domain-specific languages.

1.2 Applications of Program Generators

The two major applications for program generation techniques in the context of this work are as a *reuse technique* and as an *implementation technique for DSLs*. We now describe how program generation is suited for each of these applications.

1.2.1 Generators as a Reuse Technology

Program generators are particulary useful in the context of software reuse [11]. 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 behaviour of some parts [20].

The library developer can anticipate to 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. Maintaining and evolving such a library becomes quite a difficult task. This is because for each new possible variation, a complete part has to be written, often leading to code duplication. This library scaling problem [6, 5] hampers maintainability of the library as too much code is duplicated. Another solution is to leave the reusable part open to adaptation by the library user. Object-oriented frameworks are more flexible and are specifically designed to be customized by the application developer.

Although these reuse technologies have also proven their usefulness, the application developer is still forced to adapt the reusable part by hand and craft the required source code to fit his particular requirements. To accomplish this, the developer needs a deep understanding of the part's internals and adapting it is often not a trivial task, if it is possible at all. Furthermore, and most importantly, no means exist to guarantee that the customized framework or library part works as expected. Last but not least, in many cases, it is even impossible to package a reusable implementation as a component, subroutine or framework. Object-oriented design patterns, for example, and other collaboration schemes cannot be implemented by either of the mentioned reuse technologies without requiring specific adaptations by the developer.

This is where generative approaches are most applicable and provide a powerful reuse technology that tackles some of these shortcomings. The library provider implements a generator that is able to generate all different versions of the reusable part. The application developer can then choose among the available 'versions' to match his requirements instead of implementing them himself. A generator thus represents an entire (and closed) family of library parts: each generated part implements a different combination of functionalities offered by the generator. The application developer obtains this generated part by providing a specification to the generator. This specification is commonly written down in a kind of language defined by the generator. Since this language is commonly designed to express a particular kind of specification, it is often referred to as a domain-specific language. This is why generators are also often seen as compilers for domain-specific languages. The generator compiles a specification written in a domain-specific language into an executable program.

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 viable solution is to build a software generator that generates a parser for a given (programming) language as described by the developer. Parser generators such as Lex & Yacc [16] and SmaCC [7] have more than proven their usefulness. 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 [24] is a generator for data structures. The output of the generator is an encapsulated data container.

1.2.2 Generators as Implementations for Domain-specific Languages

As we already mentioned in the previous section, program generators are implementations for domain-specific languages. Like any other programming language, domain-specific languages are used to implement or describe software. Programming languages offer a set of language constructs with well defined semantics and a set of rules that define the valid set of programs which can be written in the language. These language constructs are the smallest program construction units. A program is thus nothing more then a valid composition of these construction units.

Software developers are trained to analyze a problem and express it in a program. The closer the abstractions of the problem domain are with the available construction units, the easier the problem can be expressed as a composition of the available construction units. In order to close this gap, a great number of different language paradigms and languages have been build by language designers. Although particular problems were more easily expressed in one programming language then the other, each language remained turing complete. Because of this requirement that every program must be expressible in every language, languages designers were forced to provide a set of generic construction units and accompanying rules. These kind of languages, often called general purpose languages, could not completely close the gap between the abstractions of the problem domain and the generic construction units.

To further close the gap between the problem domain and the solution domain, language designers were forced to give up turing completeness. This resulted in a new set of programming languages, the so-called domain-specific languages (DSLs). The language constructs of domain-specific languages completely coincide with the abstractions used in the domain of the language. Expressing a problem in terms of a valid program becomes therefore very straightforward.

Language developers soon realized that the compiler for a domain-specific language could greatly be simplified by transforming the expressions of the DSL to expressions of a GPL and subsequently compiling the latter by the GPL compiler. This strategy was quickly adopted by the majority of the DSL developers since it allowed not only the reuse of the compiler but also the reuse of the libraries, frameworks and components.

1.3 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.

Chapter 2

Program Generation Technologies

In this chapter, we present an overview of implementation techniques for program generators and domain-specific languages, based on the classification we provided in the previous chapter. For each technique, one or more existing technologies are briefly described.

2.1 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 most 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 [1].

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 enormeous 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 [16] and SmaCC (parser generator in Smalltalk) [7] .

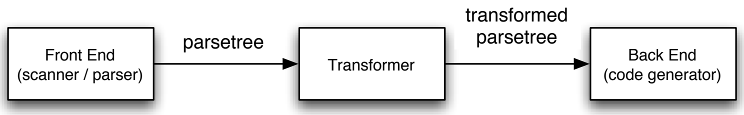


Figure 2.1: Traditional Compiler Architecture.

2.2 Metaprogramming Languages and Libraries

Instead of building a program generator from the ground up, 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) [23, 13], OpenJava [28], OpenC++ [8] and the .net CodeDOM [18] are typical examples of metaprogramming libraries available in a general-purpose programming language. They are also referred to as API-based program generators [33], as the library provides an interface that can be used to perform program generation. Another kind of metaprogramming is through syntax-extension mechanisms such as macro's, 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 build 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 advantage over ad-hoc generators is of course the reduced effort of implementation but also 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 [23, 13]. 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 overriden 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.

```
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

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.

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. For example, Lisp and Scheme macro's are much more powerful than C macro's because they operate on the program representation rather than on strings. In general, macro's are functions that are executed at compile-time and translate a part of the program in which they are used. Macro's 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, sometimes even with domain-specific syntax. A macro definition can be compared with a transformation definition, which is described in the next section. The execution of macro's at compile-time is often referred to as macro expansion, this is because macro's operate in place by transforming the syntactic language construct they define into existing, native language constructs. Therefore, macro's are the most simple kind of program transformations integrated as a metaprogramming facility in a general-purpose language.

2.3 Transformational Generators

Transformational generators constitute a large body of generators being used today. We classify them this way because 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 [29]. 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 [32].

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. Many programmers are already accustomed to the most simple kind of this generative programming technique in the form of macro's found in programming languages such as Lisp, Scheme and C. In macro's and also in the example above, 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 iinput 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.

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 transformations 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 [32]. 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.

2.3.1 Draco

Draco [20] is an approach to domain engineering using domain-specific languages and transformation technology, designed and implemented by John Neighbors [20]. 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, ... Modelling 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 modelling 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.

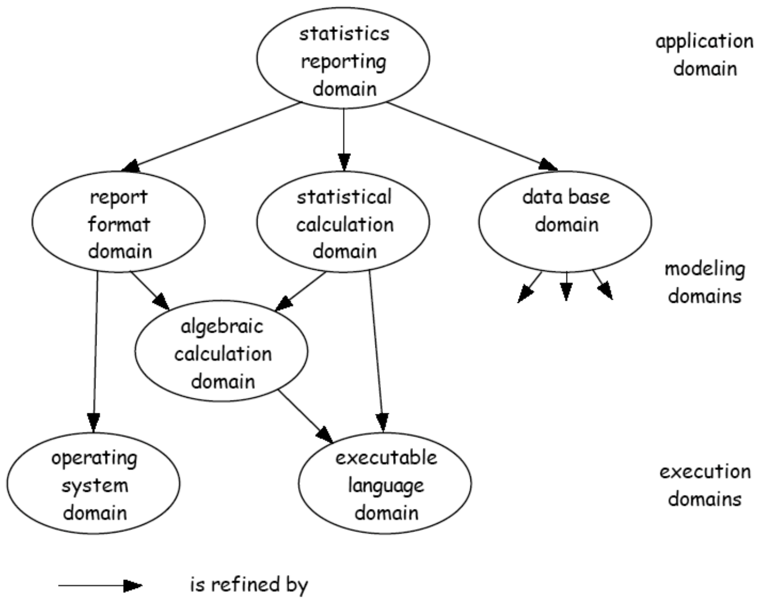
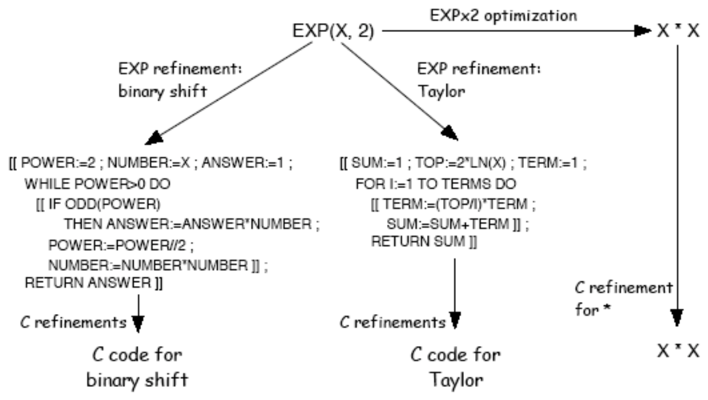


Figure 2.3: Stepwise refinement through Draco domains (from [10]).

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)
```

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.



```
[[ 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 [10]).

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 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.

2.3.2 Intentional Programming

The primary focus of the Intentional Programming system (IP) is modular language implementations [10]. 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 mistery 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 [10, 19, 2].

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 how 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 methods 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 transforma-

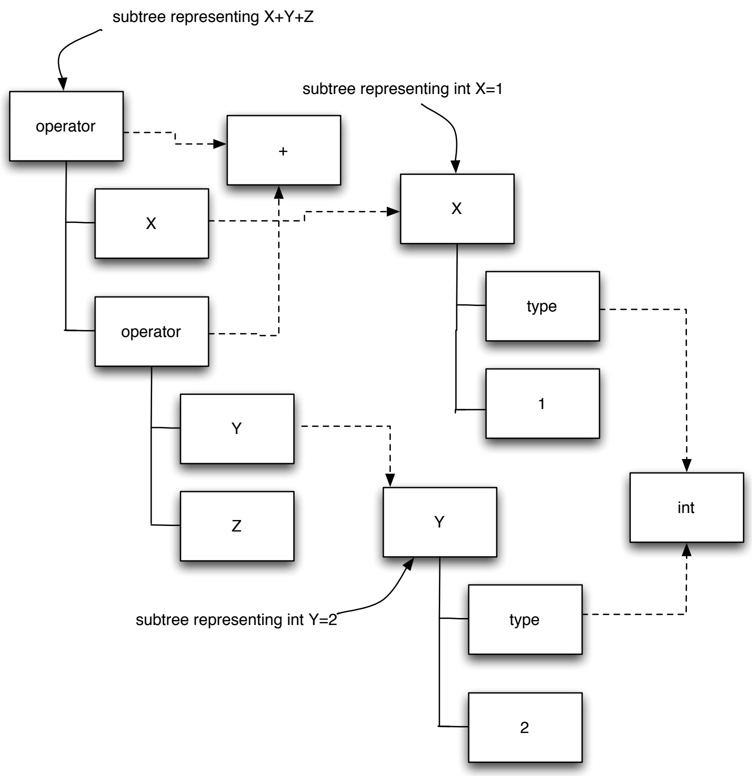

subtree representing X+Y+Z
operator
+
subtree representing int X=1
X
X
type
operator
1
Y
Y
Z
int
subtree representing int Y=2
type
2

Figure 2.5: Source tree for `x+y+z` and `int x=1`, `int y=2` (adapted from [10]).

tions. In an open system where new transformations can be introduced in the system, it is unpractical 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 neighbouring 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 neighbouring nodes should have methods that forward the method invocation to their neighbours 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.

2.3.3 ASF+SDF Meta-environment

ASF+SDF Meta-Environment[17] is an interactive development environment for the generation of interactive systems for manipulating programs, specifications, etc. ASF+SDF is the result of the marriage of two formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Defintion Formalism). The syntax is described with context-free functions of the form `symbol1 ... symboln -> result symbol`. The semantics is defined by equations written in ASF. Unlike other transformation systems there is not a strict separation between the concrete syntax and the syntax used in the equations. The concrete syntax of the language being defined can be used inside the equations for that language instead of using the abstract syntax.

Below is an example grammar specification is shown of the grammar for a small tree language using BNF syntax and ASF+SDF syntax. The language has three kind of nodes: a number `NAT`, an `f` and a `g` node. The last two nodes each contain two other nodes. Grammar specification in ASF+SDF:

```
"f" "(" Node "," Node ")" ->  Node
"g" "(" Node "," Node ")" -> Node
NAT -> Node
```

Same grammar but written in BNF:

```
Node ::= "f" "(" Node "," Node ")"
Node ::= "g" "(" Node "," Node ")"
Node ::= NAT
```

ASF+SDF also offers an abbreviated notation for a functional syntax style. The specification below contains the above grammar rewritten in the abbreviated notation:

```
f(Node,Node) ->  Node
g(Node,Node) -> Node
NAT -> Node
```

ASF+SDF is a typed rule based system. The equations are executed as rules and must be type safe. An equation consists of two open terms (AST nodes) L (left hand side) and R (right hand side), possibly containing variables. Because ASF+SDF is typed, the variables used inside terms must be declared and typed as well. By default, every function (non-terminal) tag name is declared as a variable name which is type of course the function itself. The default variables for the tree language are listed below:

```
Node -> Node
NAT -> NAT
```

To distinguish between several terms of the same type extra variable ranges are declared. In the example below, every name starting with 'Node' and followed by a number e.g. `Node1`, `Node90`, `Node342` is a variable of type Node.

```
Node[0-9]* -> Node
NAT[0-9]* -> NAT
```

For an equation to be correct, the type of the variables must match the contents of a term and all the variables that occur in R must also occur in L. Let us walk through some example equations (see below). The first one rewrites every node `f` with the value 5 into a new node `f` with the value 6. The second one rewrites every `f` node into a `g` node. Finally, the third rule rewrites every `f` node with two equals numbers to the number.

```
[1] f(5) = f(7)
[2] f(NAT3) = g(NAT3)
[3] f(NAT1, NAT1) = NAT1
```

The right hand side of an equation must always be a new term (AST node) which type corresponds to the name tag of the function (non-terminal) defining that term. So composing two existing terms in the right hand side of the rewrite rule is not possible. Consider the following _false_ language defining h-v trees.

```
context-free syntax
"h" "(" Node ")" -> Node
"h" -> Node
"v" "(" Node ")" -> Node
"v"  -> Node
"node" "(" Node "," Node ")" -> Node

equations
[1] Node(node2,node1) = node2(node1)
```

The aim of the first and only rewrite rule is to rewrite tree terms of type Node containing two other terms of type Node into the composition of first child term with the second child term respectively called `node1` and `node2`. Since ASF+SDF cannot create a new term with the type of term being another term, ASF+SDF compiler tries to parse the right hand side as a term that contains two other terms `node1` and `node2`. Luckely for us, the syntax does not correspond with the last syntax definition, and this interpretation fails.

A equation may also preceded by a condition. The equation below increments the number in all the `f` nodes by one if the number is bigger then 5.

```
[4] NAT3 > 5 and NAT4 := NAT3 + 1
    =================
     f(NAT3) = f(NAT4)
```

The search for an applicable rule is determined by the reduction strategy, that is, the procedure used to select a subterm for possible reduction. In our case the leftmost-innermost reduction strategy is used. This means that a left-to-right, depth-first traversal of the term is performed and that for each subterm encountered an attempt is made to reduce it. Next, the rules are traversed one after the other. The textual order of the rules is irrelevant. If the selected subterm and the left-hand side of a rule match, we say that a redex has been found and the following happens. The conditions of the rule are evaluated and if the evaluation of a condition fails, other rules (if any) with matching left-hand sides are tried. If the evaluation of all conditions succeeds, the selected subterm is replaced by the right-hand side of the rule after performing proper substitutions. Substitutions come into existence by the initial matching of the rule and by the evaluation of its conditions. For the resulting term the above process is repeated until no further reductions are possible and a normal form is reached (if any).

Very often rewrite rules need to traverse a particular tree or subtree in order to: retrieve information or nodes that are located elsewhere in the tree, retrieve context information, apply a rewrite rule in a particular subtree, etc. Because this is quite cumbersome to implement with rewrite rules, extensions are made to the rewrite rules.

The extension ASF+SDF provides are traversal functions [30]. There are three kinds of traversals functions transformer, accumulator and transformer accumulator. The transformer function traverses his first element and may carry extra information along its path in its other arguments. An accumulator traversal traverses his first element and updates the accumulator with the right hand side of the traversal function. An accumulator transformer traversal combines the two. The definition of such a function is similar to the other equations. During traversal the traversed nodes are matched against the first argument of the function and the other arguments given when the traversal was initiated are matched with the rest. To each of these three traversal a visiting strategy can be specified and one can specify whether a traversals stop at matching occurrences using the break keyword or continues using the continue keyword. The two supported strategies are bottom-up and top-down.

Consider for example the following `inc` traversal which increments all numbers by one. The first line is the definition function by a syntax definition followed by a traversal tuple containing the kind of traversal, the visiting strategy and the visiting policy. The semantics of the function is defined by an arbitrary number of equations.

```
context-free syntax
inc(Node) -> Node {traversal(trafo, top-down, continue)}

equations
[1] inc(N) = N + 1
```

The type system of ASF+SDF can also work against the language developer. Types are only derived from the name of the non-terminal. Because there is no subtype relationship between types, the a variable in a term must be of the exact type of the content it suppose to match. Consider the following _false_ language definition.

```
context-free syntax
  "a" -> A
  "b" -> B
  A -> Node
  B -> Node
  f(A,B) -> Node
  f(B,A) -> Node

hiddens
  variables
    "node"[0-9]* -> Node

equations:
  [1] f(node1 node2) = node2
```

In the language the first and only rewrite rule wants to rewrite the tree `f(A,B)` into the tree with one element `B` but without referring in the rewrite rule whether the first and second argument contain the node `A` or `B`, instead using the type Node. For the current language definition the rewrite rule wont work because during parsing of the expression `f(a,b)` the tree `f(A,B)` (according to the fifth syntax function (production)) does not contain any nodes of type Node.

2.3.4 XSLT

The XSLT language [29] 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 [29]) 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 explicitely 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>
```

2.4 Compositional Generators

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 wether 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.

2.4.1 GenVoca Generators

GenVoca is a design methodology for creating software product-lines. 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 [10] and Java mixin-layers [25]. 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))
application 2 = 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 behaviour 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. 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 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

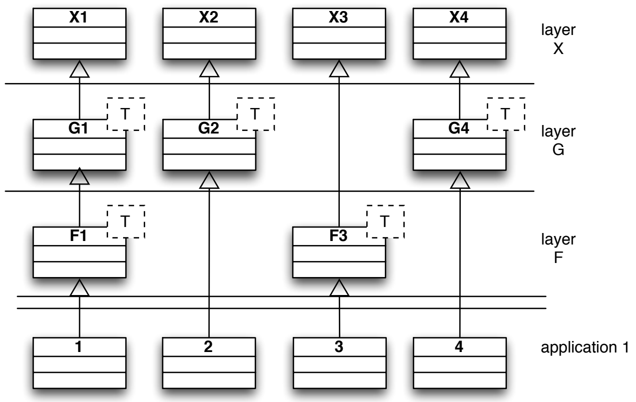


Figure 2.6: The inheritance hierarchy implementing the `f(g(x))` GenVoca equation.

of the layers, etc....

2.4.2 Subject-oriented Programming and Multidimensional Separation of Concerns

In Subject-Oriented Programming (SOP) [21, 15], 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 multi-dimensional separation of concerns (MDSOC) [22], 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

DSL Evolution Problems

Domain-specific languages that are implemented using program generation techniques, and more specifically using program transformation, are also prone to evolution problems. In this chapter, we identify those issues that need to be solved to enhance evolvability of DSL implementations.

3.1 Introduction

Despite the obvious advantages of DSLs, the disadvantage is that because the underlying libraries, frameworks and components written in GPL evolve, the DSL compiler must evolve as well. Over time, new functionality is added and new notions in the domain are introduced, which must be incorporated in the DSL.

The cost of building a full implementation for a DSL compiler from scratch has been greatly reduced by the advent of transformation systems. Unfortunately the cost of adapting an existing DSL implementation or the implementation of another similar DSL often boils down to the same amount of work. Naturally, given the already existing implementation of a DSL, this cost to high. The reuse of parts of a DSL implementation gets limited by the high degree of coupling between those parts. Whenever a part evolves or changes, the other parts get invalidated or even corrupted. In this chapter we investigate what causes the high degree of coupling between these modules and what the effects are on future evolutions of the language.

3.2 Compiler Design

The construction of a compiler for a DSL is based on the architecture of a compiler for general purpose languages. The architecture of such compilers divide a compiler into several functional layers. The most common layers are lexing, parsing, several transformation layers like type checking, transforming, optimizing, finally pretty printing etc. The layers are put on top of each other so that the output produced by a layer serves as the input for the next layer. A transformation layer consists of a set of transformation modules each dealing with a specific part of the transformation.

Usually BNF or some variant is used to specify the DSL grammar, but of course other formalisms can be used as well. From the grammar a parser can be constructed. This can be done very easily and a lot of tools like JACC, SMAK, etc are available that automate this process. After the parser is released on the code text an syntax tree (AT) is produced. The AT is pruned from concrete syntax information, since this information is not relevant for the rest of DSL implementation. The result of this pruning is an abstract syntax tree (AST) which is called the source AST since it is derived form a DSL program. The AST is then processed by several transformation layers which manipulate the tree until a new and valid AST has been obtained. This AST is the AST of the target program written in the desired target language. We will refer to this tree as the target AST. Depending on the transformation system at hand, this tree is immediately handed over to the compiler of the target language or is pretty printed into a character stream which is then given to the compiler of the target language.

In the layered architecture of the compilers, the output of one layer serves as the input for the next layer. In order for this scheme to work, the structure of the input accepted by one layer must be carefully tuned to the output of the previous layer. This dependency between two layers burdens and jeopardizes the evolution because changes in the output of one layer can cause various errors in the successive layers.

The kind of errors and their traceability depend on the specific paradigms and implementations of the transformation we can identify 3 kinds of problems that may occur:

failure During the execution of the transformation module, the modules or the whole layer breaks.

invalidate The transformation module apparently still works but the module returns a different result.

ineligible The transformation module is no longer triggered because the conditions for the module are no longer satisfied. When the module is not removed from the compiler, it is dormant and may become active again and cause unexpected results.

Although this a architecture that fits most transformation systems, note that the different architectural layers may not always be cleanly separated and presented to the developer of DSL compilers. However, we will use this architectural schema in our analysis of the problems regarding the evolution of DSL implementations with transformation systems.

3.3 Transformation Modules

3.3.1 Identification

There are conceptually two viewpoints for modeling the transformations source driven and target driven. The viewpoints respectively takes the source language and the target language as the dominating structure to determine the identification of the transformation modules, the scheduling of the transformation modules and the composition of the results of the transformation modules. In the source driven viewpoint or architecture, transformation modules are identified and constructed based on particular parts of the source AST. The default scheduling mechanism follows the structure of the source language ranging from leftmost-innermost to rightmost-outermost but can be customized, if desired. The results of the transformation modules are composed according to the structure of the source AST nodes that are transformed. Transformation systems that follow this viewpoint are ASF+SDF, TXL, XSLT, BCG etc. In the target driven viewpoint or architecture, transformation modules are identified and constructed based on the desired parts of the target AST. The scheduling and composition is entirely based on the target structure and must thus be specified by the user. Transformation systems that follow this viewpoint are LMP, Joistra, Templates, XSLT, ... etc.

Because the transformation modules in these two viewpoints are oriented differently, languages and their non-functional requirements like reuse and evolvability are conceived also differently. In the source driven viewpoint new languages are designed, constructed and implemented by a set of source language constructs. Each language construct is transformed by a number of transformation modules. Existing languages evolve through new language constructs and by refining and enriching the language constructs semantics.

In the target driven viewpoint new languages are designed, constructed and implemented by a set of desired targets. Each target is constructed by a number of transformation modules which retrieve their information from the input AST (often called a model). Existing languages evolve through new targets and by refining and enriching existing targets.

In order to further discuss the evolution and reuse problems of those two viewpoints in general we use the syntax of two representative transformation systems. Currently, only examples in the source driven viewpoint are given. The representative for this viewpoint is ASF+SDF.

3.3.2 Granularity

As in any programming model, the granularity of the entities that are constructed is very important. Coarse grained entities are usually more straightforward to build, because less effort is needed to decompose the problem into a set of finer grained entities. On the other hand such entities are usually hard to evolve and to reuse. When writing a set of transformation modules, decisions on the granularity of those entities must also be made .

Let us illustrate the two sides of the granularity spectrum to implement a DSL compiler with rule-based transformation systems. Consider the following AST of a program consisting of one subject which contains an article and a title. A title has two parts, a main title and a subtitle.

```
subject(title("first title", "second title"),
       article("The new council abruptly canceled ...")
```

Suppose we want to compile this AST into a HTML page. The titles must be centered and printed on top of each other at the top of the page. The subtitle must be printed in gray and in font size 9.

In the first approach the rewrite rules match the largest possible structure which they still can rewrite. The advantage of this strategy is that the implementation of a transformation is very simple. This is certainly the case for this example, because the whole transformation can be written using one big rewrite rule (shown below). The rule matches a subject containing a title and rewrites this tree into the required HTML code.

```
context-free syntax
   font(element*, int, string) -> element
   cap(element*) -> element
   "br" -> element
   center(element*) -> element
   text -> element

   subject(title article) -> subject
   title(text,text) -> title
   article(text) -> article

equations
   [1] subject(title(text, text2), article(text))
            = center(text br font(text2, 9, "gray")) br text
```

The first disadvantage of this strategy is its sensitivity to any increase in variability of the language. Any slight increase will most likely force existing rewrite rules to be split up hence, introducing a lot of duplication. Let us consider two small changes. Suppose we extend the language (shown below) so that a subject contains a title and a set of articles. Rewriting the articles in the subject rewrite rule would require us to iterate within a rewrite rule over different articles, something which is not possible in rewrite rule systems. So we are obliged to write down another rewrite rule (line 2). Note that this is not a shortcoming of rewrite rule systems. Adding this kind of iterative capabilities would break down the rewrite rule system to yet another general purpose language.

Another disadvantage is that a lot rewrite logic gets duplicated. Because of the duplication, future evolutions will most likely require changes in several rewrite rules. Consider another language extension (also shown below) that introduces a variant of a title that only consists of a main title. Rewriting both kinds of titles in a single rewrite is simply not possible. We must copy the subject rule and adapt it so that the rule handles the new kind of titles (3). In the process the transformation logic concerning the main title and the subject got duplicated.

```
context-free syntax
   title(text) -> title
   article(text) -> article

equations
   [1] subject(title(text, text2), articles*)
            = center(text br font(text2, 9, "gray")) br articles*

   [2] article(text) = text

   [3] subject(title(text), articles*) = center(text) br articles*
```

In order to avoid those evolution and reuse problems, it is clear that we need to take another approach. The only other strategy is to implement the transformation by dividing it in small rewrite rules that capture the smallest possible structure which they still can rewrite. This strategy maximizes the amount of reusable rewrite logic and minimizes the need to split up existing rewrite rules. Any increase in variability requires a minimum amount of implementation effort because of the reuse possibilities and avoids constant refactoring of the existing rewrite rules.

The example program contained two concepts: a title and a subject. The implementation of the transformation according to the new strategy yields (see below) in a rewrite rule for titles (3), a rewrite rule for the article (2) and one for subjects (1). The title is rewritten into a html structure rendering the main and the sub title centered, in the requested font and on top of each other. Since the subject merely contains a title, the subject is rewritten by its title with a subtitle.

```
context-free syntax
   subject(title) -> subject

equations
   [1]  subject(title, article) = title br article

   [2] article(text) = text

   [3] title(text,text2) = center(text br font(text2, 9, "gray"))
```

The two small evolutions can easily be incorporated in this implementation of the language. Adding articles to the subjects merely involves changing the subject rewrite rule to incorporate more then one article (1) and adding a variant of text requires only the addition of one new title rule (2).

```
[1] subject(title articles*) = title br articles*

[2] title(text) = center(text)
```

Although this fine-grained division of a transformation into several rewrite rules works perfectly well in this example, the strategy has also several drawbacks. Each of these drawbacks is discussed in the following sections. Firstly, the strategy only minimizes the dependencies between de grammar and the transformation and vice versa but not all of the dependencies can be completely eliminated. The remaining dependencies are discussed in section 3.4. Secondly, because of the transformation on transformation dependencies, such transformations can never be completely written separately (section 3.5). Thirdly, transformations that only my execute in a particular context or need information out of their context get broken by this strategy (section 3.6).

3.4 Grammar and Transformation Dependencies

The specification of the grammar and the implementation of the transformations are separated in different layers. This separation allows the two parts to evolve independently. Besides the benefits like multi-target transformations for example, the danger of this approach is that the grammar and the transformation can evolve out of synch: transformation layers can impose new constraints, can remain ignorant of certain features of the grammar or may even be corrupted because of changes to the grammar.

The first transformation layer takes as input the source AST from the parsing phase. Each transformation module in the transformation layer takes a portion of the source AST and produces a piece of the target AST. To avoid any discrepancy between the grammar's output and the transformations input one must carefully check wether all information captured in the grammar is used by the transformations and whether the assumptions on the structure made by the transformations are met by the grammar. These two sanity checks must be performed whenever the grammar or transformation systems are changed.

3.4.1 Assumptions on the input AST by the transformations

The rewrite rules of any rule-based systems consists of a left hand side and a right hand side. The left hand side is a pattern which is matched against the source AST. In order words, the left hand side of each rewrite rule is an assumption on the structure of a particular part of the source AST. The complete set of these assumptions dictate the complete structure of the source AST. The consequences or severeness of what can wrong when one of the assumptions are not met by the source AST of the grammar depend on the semantics of the rewrite rules.

Whenever the patttern of a rule matches a set of nodes of the source AST, the top level node of that match is replaced by its right hand side. The right hand side consists soley of either new elements that must be created or elements that have been matched by the left hand side. Therefore rewrite rules cannot fail during their execution or become invalidated. In this paradigm rewrite rules either match or they do not match, hence rewrite rules can only become ineligible. Let us illustrate this kind of error when one the assumptions is not met by the grammar. In the example a set of rewrite rules are given for a initial grammar where charts and articles are siblings in a larger structure. Because common parts of charts and articles must be treated differently, the set of rewrite rules implicitly assumed that charts are not contained in articles. Next, an seamless innocent evolution of the grammar is presented which allowed charts to be included inside articles. Because the assumption within the rewrite rules was no longer satisfied certain rewrite rules become ineligible.

Consider the following AST which describes a multimedia article of a newspaper. Articles are contained in a subject, accompanied by a title and a set of multimedia objects.

```
(subject OilPrice
   (title "Index Page")
      (article (text "The amount of oil produced by the OPEC"
                     (comment "OPEC ..." ) "..." ))
         (chart "Oil Prices of 1999 until 2003"
            (x (range 1999 2003))
            (y (value 10)
               (value 20)
               (value 30)
               (value -40 (comment "War in Irak") ))
```

Each subject corresponds to a separate page. At the top of the page the title of the article is printed followed by the articles text and was concluded by a list of multimedia objects. Although both the article text and the graph contain `comment` nodes, the rendering of the comments depends on their context. Comments in articles are displayed as three little dots. A mouse click on the dots is required to be able to read the comments. Comments in charts are immediately shown when the mouse points to the a particular value on the chart.

A set of rewrite rules were developed in order to transform the above source AST into a HTML page. The following set is a snapshot of the entire set of rules that only deal with comments. First the `subject` tree is rewritten into the tree `transform_chart(transform_article(Subject))` and immediately after the substitution the newly created tree gets first of all rewritten triggering the `transform_article` rule and afterwards the `transform_chart` rule. The `transform_article` rule traverses the subject until it reaches the `article` node, which triggers its execution and so forth. This set of transformation rules will thus first traverse the article and subsequently the chart.

```
Subject = transform_chart(transform_article(Subject))

transform_article(Article) = transform_article_comments(Article)
transform_article_comments(Comment(Text))
   = <a href="javascript:ShowInWindow(Text))"> "..." </a>

transform_chart(Chart) = transform_chart_comments(Chart)
transform_chart_comments(Comment(Text))
   = <a onmouseover="javascript:popup(Text))"> "..." </a>
```

Consider the following evolution of the example language where an article also may contain inline multimedia objects. After changing the grammar to reflect this need, we could write the following example AST in which the graph is now directly contained in the text of the article.

```
(subject OilPrice
   (title "Index Page")
      (article
         (text "The amount of oil produced by the OPEC"
            (comment "OPEC ..." )
            "..."
            (chart "Oil Prices of 1999 until 2003"
            (x (range 1999 2003))
            (y (value 10)
               (value 20)
               (value 30)
               (value -40 (comment "War in Irak") ))))
```

The semantics of the language wasn't changed so we should be able to use same set of rewrite rules. We would also expect that the outcome of the transformation is the same. But this isn't the case. Running the set of rewrite rules against the above example results in a graph with comments represented as dots instead of popup windows. When we take a closer look at the rewrite rules we discovered an implicit assumption. During the transformation of the article, every comment contained in the article is changed. Afterwards, the charts and their comments are transformed. This implementation worked because it assumed that the chart wasn't a part of the article. The change in the grammar, permitting us to put also charts inside the articles, placed the charts hierarchically under the article node. Therefore also the graph comments got rewritten as they were normal article comments.

3.4.2 Assumptions on the transformation system by the grammar.

The left hand side of a rewrite rule is a pattern that is matched against the source AST. The total set of these patterns determines the parts of the source AST that are taken into account during the transformation process. Naturally one expects that all the information captured by the grammar is taken in to account by the transformation process. The only way to make sure that all the information of the grammar is properly used is to check wether every AST structure that is allowed by the grammar is matched by at least one rewrite rule. For large and complex systems, this is a complex and cumbersome task.

Grammars are a very powerful means to describe the set of valid expressions of a language. Their expression power becomes particularly visible in cases where grammars need to capture a large number of combinations. To capture a number of combinations we only rely on two mechanisms: alternatives and references to other productions. Each production which refers to other productions define a combination. The alternatives for the referenced productions are declared by other productions. The number of combinations is the product of each number of alternatives for every referenced production. Capturing a number of combinations in grammars is so natural we often even don't realize the total number of possible combinations.

While capturing a large number of combinations in the grammar can be very declaratively expressed. The transformation of ASTs derived of such grammars must be able to deal with every combination or group of combinations individually. Even with a systematic development of the necessary set of rewrite rules, certain combinations allowed by the grammar get easily overlooked. Let us illustrate this kind of problem.

The initial DSL used in the example is a DSL for describing games consisting of objects and rules telling how the objects should interact. The semantics can expressed using only two rewrite rules. Next we propose an evolution of the language where we add support for active objects, these are objects that move by themselves. The extension of the language involves merely a simple alternative of one of the productions of its grammar, but it tripled the number of rewrite rules needed to implement its new semantics.

Consider the following AST which describes a game consisting of two objects: a car and a wall, and one rule which is triggered when the car hits the wall.

```
(rule
   (normalobject(CAR))
   (hit)
   (normalobject(WALL))
   (...))
```

The language of the AST consists of three concepts: objects, actions and rules. Objects are graphically rendered on the screen and can be moved to any valid position on the screen. Actions denote a relationship between two objects. Rules state which relationships between two objects must hold. When the relationship holds, the code attached to the rule is executed. The first three lines below, define an object, the hit action and under action. The next 4 lines define a game rule consisting of two elements, an action and a piece of code which is executed when the rule is triggered. In the last part, the semantics of the language is specified. Note that there are only 2 rules needed, one rule to handle hit actions and another rule to handle the under action. The hit action defers from the under action because when hit actions occur a specific event object is returned that can be used to retrieve properties of the event.

```
concrete syntax
   "object" "(" ID ")" -> object
   "hit" -> hit
   "under" -> under

   "rule" "(" element "," action "," element "," code ")" -> rule
   object -> element
   hit -> action
   under -> action

equations
   [1] rule(normalobject(ID), hit, normalobject(ID2), code) =
   [2] rule(normalobject(ID), under, normalobject(ID2), code) =
```

Till now, the language only supports passive objects which can only be moved by user input. But in many games certain objects move by themselves according to a certain pattern. Therefore the language is extended by a new kind of objects, called moving objects. These objects have their thread of execution and compute their own position on the screen. When a rule with moving objects is triggered, the motion of the moving objects must be suspended before the code of the rule is executed and must afterwards be resumed again. Using these kind of objects, we can write the following AST of a game. It consisting of two objects: a bouncing ball and a wall, and a rule which is triggered when the bouncing ball hits a wall.

```
(rule
   (movingobject(BOUNCING_BALL))
   (hit)
   (normalobject(WALL))
   (...))
```

Extending the grammar with moving objects requires one production specifying the syntax of a moving object (first line) and requires an alternative for the element production (second line). Because of the ease of being able to add another possibility for an element of a rule, we are not confronted with the huge increase of new possibilities that are now captured in the grammar. The rule production refers to the action production and refers two times to the element production. Multiplying the number of combinations of an action with the square of the number of elements results now in 8 possible combinations, that is 3 times the number of possibilities captured in the original grammar. Since normal objects and moving objects must be treated differently, these 6 new combinations (last 5 lines) must be captured by individual rewrite rules. First we need to determine which are the new

6 combinations compared to the original implementation. Then we need to determine for which of these new combinations a new rewrite rule is needed and subsequently add it to the set of rewrite rules. In our example we only needed 5 rewrite rules.

Even the extension of this little language with one alternative resulted in an increase of a large number of combinations. The extension of this language with various unary and binary actions, object layouts, input-devices, etc. towards a full fledged gaming language can rapidly increase the number of new combinations. During the review of the existing set of rewrite rules to determine the new combinations and while determining the necessary rewrite rules, certain combinations may easily get overlooked.

```
concrete syntax
   "movingobject" "(" ID ")" -> movingobject
   movingobject -> element

equations
   rule(movingobject(ID), hit, movingobject(ID2), code) =
   rule(movingobject(ID), hit, normalobject(ID2), code) =
   rule(normalobject(ID), action, movingobject(ID2), code) =

   rule(movingobject(ID), under, movingobject(ID2), code) =
   rule(movingobject(ID), under, normalobject(ID2), code) =
```

3.5 Transformation and Transformation Dependencies

In a layered compiler architecture there are often several transformation layers involved, because a single transformation module was not able to transform the source AST directly into a correct target AST. The first transformation layer takes as input the AST that is produced after parsing the DSL program, the remaining transformation layers take as input the result of the previous layers. Those remaining transformation layers are particularly difficult to mange because their input structure is no longer well described as opposed to the input of the first transformation layer. Maintaining the consistency between grammar and first layer was hard, one can imagine that the maintenance of the next layers will only increase.

When a transformation of at least two source AST elements needs the result of the transformation of another AST element then the former transformation must be divided in several modules and put in several transformation layers. These transformation modules will then transform the results of previous transformation modules until the desired target AST is obtained.

Consider the following AST of a program consisting of one subject which contains a two titles: a main title and a subtitle.

```
subject(title("first title", "second title"))
```

Suppose we want to compile this AST into a HTML page. The titles must be centered and printed on top of each other at the top of the page. The subtitle must be printed in a font size so that the width of the title and subtitle are the same. Additionally when the subtitle is a title of a subject, the subtitle must also be printed in gray.

The example program contained two concepts: a title and a subject. The implementation (see below) yields in a rewrite rule for titles (3) and one for subjects (1).

```
concrete syntax
   font(element*, string) -> element
   font(element*, int) -> element
   cap(element*) -> element
   "br" -> element
   center(element*) -> element
   text -> element
   subject(title) -> subject
   title(text,text) -> title

   spraygray(element*) -> element { traversal(trafo, top-down, break) }
   subject(element*) -> subject

equations
   [1]  subject(element*) = spraygray(element*)

   [2]  spraygray(font(text, int)) = font(font(text, int), "gray")

   [3]  size := computefontsize(text,text2)
        ====================================================
        title(text,text2) = center(text br font(text2, size))
```

The title is rewritten into a html structure rendering the main and the sub title centered, in the requested font and on top of each other. Since the subject merely contains a title, the subject is rewritten by its title and its gray subtitle. The graying of the subtitle is performed by the spraygray rewrite rule which is triggered when the subject is rewritten. The spraygray rewrite rule may not rewrite just any text but only the subtitle text. Therefore the rule traverses the result of the title and is triggered when it matches a font node. Since only the subtitle is wrapped in a font node, we can make sure that the spraygray rule only rewrites the subtitle text.

As you can see from the above implementation, when transformations operate on the results of other transformations their implementation heavily relies on the results of the previous transformation. Now suppose some time later, that the main title must be printed in font size 12. The title rewrite rule is looked up and changed accordingly:

```
[3]  size := computefontsize(text,text2)
      =========================================================
      title(text,text2) = center(font(text, 12) br font(text2, size))
```

When the DSL compiler is run against our input, the main title will be printed in gray as well. Unwillingly we broke the implicit condition of the capitalize rule, which is now unable to distinct between the text of the subtitle and the text of the main title because both parts of the title are contained in a `font` node.

One might suggest that another way of implementing this transformation is by first traversing the source AST and rewrite the subtitle text by a grayed text and subsequently rewrite the title. In this example, this idea does not resolve the problem. Consider the alternative implementation here below:

```
[1] subject(title) = spraygray(title)

[2] spraygray(title(text,text2)) = title(text, font(text, "gray"))

[3] element2 = font(text2, string)  and
    size := computefontsize(text, text2)
    ==========================================================
    title(element1,element2) = center(text br font(element2, size))
```

The title rewrite rule needs to access the text of the subtitle in order to compute its fontsize. Therefore the subtitle (`element2`) in the rule is matched is against the font node, rendering it dependent on the spraygray rewrite rule. We thus end up with the same dependency.

There are of course other examples, where this idea would work but it would only shift the problem. Instead of making the rewrite rules dependent on target results produced by other transformations, the rewrite rules would depend on a intermediate structure that is no longer equal to the structure described the grammar.

3.6 Information dependencies

A transformation transforms the source AST structure into an the target AST structure. Naturally, all the necessary information to construct the target AST can thus be retrieved from all over the source AST. As we explained in the previous sections, a transformation is implemented by a set of transformation modules. The division into transformation modules assigns to each transformation module a responsibility to construct a part of the target AST using a part of the source AST. However the part of the source AST required often consists of several source AST nodes which are not necessarily direct connected (e.g. parent child relationship). These transformations modules are called *multi-source* transformations modules or *global-to-local* transformation modules. Usually one source AST node is chosen as the pivot node from which all other nodes can be reached. This choice mainly depends on the specific capabilities in transformation systems to reach other nodes given a node. When the context of a node can not be reached the most top-level node is chosen, otherwise the triggering node is usually chosen. The other nodes necessary nodes are called *n*on-local nodes. Transformation modules that use the pivots to retrieve the non-local nodes introduce dependencies between the elements of the source AST structure. These are called the information dependencies. These dependencies serious complicate the implementation and the future evolutions of the entire language. Note that we do not distinguish between the source driven or the target driven viewpoint. Although depending on the architectural viewpoint on transformation modules the division is initiated differently, the information dependencies remain the same.

3.6.1 Cause and illustration for the need for non-local AST nodes

The language constructs of DSLs are based on the concepts of a particular domain. The language constructs allows us to use and state properties over domain concepts. Expressions in DSLs are declarative statements over the relationships between those domain concepts. In contrast, general purpose languages (GPL) on the other hand consists of a set of primitives to express state, computation referring to other computations and control flow. Clearly the structure of a DSL can significantly differ from from the structure of GPL. A compiler for a DSL must transform the declarative statements of a GPL into a set of referring computations and control flow. In the process, it must overcome the structural differences between both languages. Because of these structural problems some transformation modules need several indirectly connected AST nodes.

In order to illustrate this, consider the following DSL of which the grammar is given below. The DSL consists of general elements like a string, a set of images, a link or a reference and three main concepts: menus, pages and sections. Menus have two layers: main menus and submenus and four rendering strategies: layout for the menu captions consisting of images, and the kind of animation and layout directions for the main and submenus. Furthermore, the caption of the main and submenus is specified by a general element. Pages have a title and a arbitrary number of sections which consists of a name and a title and may also contain a series of sections, elements. Links and references either point to pages or to sections within pages (provided that the name of the section is unique).

```
root(menu,page*) -> root
menu(mainmenu*,menulayout,menulayout,imagelayout,menuanimation) -> menu
main(ID,element, submenu*) -> mainmenu
sub(ID,element) -> submenu
horizontal -> menulayout
vertical -> menulayout
toggleimage -> imagelayout
slide -> menuanimation
"vertical" -> vertical
"horizontal" -> horizontal
"toggleimage" -> toggleimage

Page(ID,title,element*) -> page
Section(ID,title, element*) -> element
title(String) -> title

ref(ID) -> element
String -> element
Image* -> element
Link(element, ID) -> Link
Image(ID) -> Image
```

The following AST of a program is written in this language. The program defines a menu and a news page. There are two main menus: headline and themes. The caption of the headline menu item is a set of images and the menu item refers to the headline section. The second main menu item is themes. This menu item expands into a submenu with two items. The main menus are horizontally layouted whereas the submenus are vertically layouted, both are animated with the slide effect. Furthermore, the images of the menu captions are toggled when the mouse hovers over the menu item. The news page consists of a title and two sections. The title of the headline section is "Explosion kills..." and refers to the iraq section.

```
menu(
   (main(headline, link( imageset((imag1,imag2)), headline, ())),
      main(themes,"Theme",
         (sub(themes,"Culture"),
          sub(themes,"Science"))),
          imagelayout(toggleimage),
          menulayout(horizontal),
          menulayout(vertical),
          menuanimation(slide)
)
page(news,
   title("News"),
      (section(headline,
         title("Explosion kills .."),
         (text(" "),
         ref(iraq)),
         ),
      section(iraq
         ...
      ))
)
```

The compiler for this language must transform such specifications into HTML. Besides the semantics described above, there are some additional requirements. The titles of the sections must be numbered and the references to the section must contain the number and the title.

The division of this transformation according to the source driven viewpoint into smaller transformation modules yields to a lot of transformation modules that require several indirectly connected source AST nodes in order to produce the desired target AST parts. Because these are the kind of transformation modules, only these modules are described in figure 3.1. Note that these descriptions are not correct ASF+SDF rewrite rule nor adhere to any other transformation system, the descriptions are written in a kind of pseudo-syntax for multi-source transformations. For each transformation module the triggering AST node is chosen to function also the pivot node. The non-local nodes required by the transformation modules are prefixed with a "+". The relationship between the pivot node and the other nodes isn't expressed because this would clutter up the rewrite rules to much. Let us quickly go over each of them:

- The rearrange menu transformation module (1) must rearrange the larger part of its arguments, making it dependent on the specific nature and combination of those arguments. The modules animates its menu and applies the layouts for the main menu and the submenus. Notice also that imagelayout is not used because a menu does not directly need it.

- The image layout transformation module (2) requires an set of images in order to implement the toggleimage effect.

- The link transformation module (3) constructs a html link to the given section hereby needing the page containing this section to complete its reference. This of course give rise to the section transformation module (7) that creates an html anchor tag around every linked section.

- The section transformation module (4) was a bit harder to write down. The module must prefix the title with its hierarchical number. Therefore all the parent sections are needed in order to construct the number.

- The ref transformation module (5) is quite similar to the section module (4) and the link module (3). The ref module constructs a html link to the section. The only direct information available for this rule is an identifier of the section. Like the link module, we need to retrieve the page containing the section and the section itself. The section is also needed because its title is part of the caption of the link. Additionally we need all the parent sections in order to construct the section number, just like the section module (4).

- The title transformation module (6) animates the title of a page just like the animation of the menu linking to it.

3.6.2 Current implementation approach

Implementing multi-source transformation modules is quite some work. Moreover a systematic approach is certainly recommended to keep an overview of the dependencies between the implemented transformation modules.

The key issue in the implementation of such transformations is the assignment of the responsibility to retrieve the non-local nodes. There are roughly two strategies: (1) transformation modules retrieve their own non-local nodes starting from their pivot node and (2) separate transformation modules transform the source AST so that all non-local nodes become available in the pivot node of the actual transformation module.

The first strategy is often quickly discarded because the transformation modules heavily depend on the location and properties of the required nodes. These dependencies hamper the future evolution of the whole compiler (cfr section 3.4). Therefore the second strategy is chosen. In this strategy the retrieval of the other AST nodes involves 6 steps:

1. **common ancestor** Find a source AST node that is a common ancestor of the pivot node and all other non-local nodes. This node serves as a starting point.

2. **find** Find all the non-local nodes starting from the common ancestor.

3. **collect** Collect the non-local nodes.

4. **distribute** Propagate the collected nodes to the pivot node.

5. **add** Add the collected nodes to the pivot node.

6. **cleanup** Prune the AST nodes of the collected nodes that are no longer needed.

In the above cookbook recipe, separate transformation modules must traverse the source AST to find the necessary nodes, and must distribute them to the pivots. Such traversals are directly supported by the larger part of the transformation systems, which were extended with traversals or similar capabilities. With a traversal over a tree all its node can be visited, information can be accumulated and/or changes to certain nodes can be made. They alleviate the developer of manually walking down the tree, talking into account every kind of node. The number of types of nodes roughly correspond to the number of productions in the grammar. Because real life languages are described by grammars that consists of a few hundreds of productions, even a simple traversal definition would rapidly become very large.

Let us illustrate with the rearrange main transformation module rule how the source AST can be rewritten using traversals. Generally, according the cookbook there are 4 groups of traversals needed: one for initiating the traversals at the common ancestor, one for finding and collecting the non-local nodes, one for distributing these nodes to the pivot, and finally one for pruning the source AST.

The implementation of the information retrieval for the rearrange main menu transformation module can be found in figure 3.2. The pivot of the module is the `menu` node. The module additionally requires all the submenus of the main menu. The common ancestor of the menu and the submenus is the menu source AST node. This node serves as the starting point for the provide_submenus (1) traversal. The submenus can be found in the main menus, hence the collect_submenus (2) traversal is invoked around the `mainmenu*`. The collect_submenus traversal matches all the `sub` source AST nodes (first parameter). An additional parameter `sub*` serving as an accumulator for the obtained submenus is added to the collect_submenus rule. In this example, there is no need to distribute of the submenus because the pivot is the same node as the common ancestor. The addition of submenus to `menu` AST node involves merely the creation of an new `menu` AST node containing the desired results. Finally, the submenus must be pruned from the `mainmenu` AST nodes, provided of course that the submenus are no longer needed. This is accomplished by the cleanup traversal (3).

3.6.3 Problems with the current approach

The support for tree traversals has greatly improved and reduced the amount of code needed to express the traversal. But when looking at from an evolution perspective, traversals do not always seem fit for the job. The traversals transform the source AST to alleviate the actual transformation modules of this task. In other words, the actual transformation modules are put in another layer taking as input a transformed source AST. In section 3.5 we described already some general evolution problems of layering transformation modules on top of each other. Beside the layering problems also the implementation of the traversals hamper future evolution. Let us analyze their implementation more in depth following the steps of the cookbook recipe.

Collect

As long as the non-local nodes do not depend on each other, each node can be collected and provided separately. But when there are non-local nodes that depend on the context of other non-local nodes then the collection and distribution become rapidly very complex. The context dependent nodes must be collected together with the other nodes on which they depend. Both nodes be stored in an intermediate node, in order not to lose their context relationship. Upon distribution the correct set of context nodes can then be retrieved together with the other nodes on which they depend.

In our example language, this situation occurs in both the title transformation module and link transformation module. Let us illustrate the above mechanism with the title transformation module.

The title transformation module animates the title of a page using the same animation effect as the animation of the menu linking to that page. So the title AST nodes lack the source AST nodes containing the animation of the menu that is linking to that page containing the title. In other words, the animation of the menu passed to the titles of the pages must adhere to the condition that the menu links to the page containing the title. The condition cannot be checked upon the collection of `animation` nodes, because during the collection phase we don't know yet for which page to look for. Only during distribution when the page titles are visited we know for which pages we need the animation. So during the collection phase, we must both collect the menu animation and the set of pages that are linked from the menu. The set of pages can then be used during the distribution to select that `menu animation` source AST node which menu links to the page containing the title.

Let us quickly run over the implementation (see figure 3.3). The common ancestor of the page titles and menus is the `root` source AST node. It is the starting point for the provide_titleanimation traversal (1). The collection of the menu animation and the set of pages that are linked form the menu is divided in three traversals. The collect_titleanimation traversal is triggered on menus where it combines in a new `pageanimations` node: the pages linked in the main menu and in the submenu with the animation of that menu. The combination into a new node was necessary because a accumulator traversal can only return one result. The collection of the menu animation is performed in traversal (3) and the collection of the pages (4) is performed in traversal. Finally the `pageanimations` are propagated to the pages where the appropriated animation is selected based on the list of page identifiers.

Distribute

The distribution of extra source AST nodes to a pivot node must be added to the compiler with caution. Different distributions may overlap because they are triggered on the same kind of nodes within overlapping subtrees. In such cases, the outcome of the compiler is totally based on the scheduling of the traversals since the scheduling prioritizes the distributions. One can of course try to refine both distributions so that their subtrees no longer overlap. But this option only shifts the problems. Instead of carefully scheduling the traversals, now upon each addition of a traversal refinements to all other traversals must be considered.

To illustrate the problems caused by these dependencies let us analyze the implementation (figure 3.4) of the image layout transformation module ((2) in figure 3.1). The module required a set of images to apply its layout upon. The pivot of the module is a set of images and the extra node is the imagelayout of the menu containing this set of images. The common ancestor of the pivot and the extra node is the `menu` AST node. The provide_imagelayout traversal (1) is triggered on the `menu` AST node. It first collects the imagelayout of the menu (2) and afterwards distributes the layout to every set of images within that same menu (3).

Suppose the language is extended with an optional image layout for the captions of the submenus. If there is no such special layout then the image layout for the main menu is used. Below is implementation of the traversals to provide the optional image layout to the captions of the submenus. The image layout of the submenu may only be distributed to the submenus, therefore the distribution is divided in two traversals (4) and (5). The first traversal is triggered when a submenu is matched. Inside the submenu, the fifth traversal is started. The latter is triggered when the set of images are matched where the traversal composes the images with the collected layout.

```
[1] layout := collect_imagelayout2(menu, ) and layout != nil
    ===========================================
    provide_imagelayout(menu) = distribute_imagelayout(menu, layout)

[2] layout := collect_imagelayout(menu, )
    and nil == collect_imagelayout2(menu, )
    ===========================================
    provide_imagelayout(menu) = distribute_imagelayout(menu, layout)

[3] collect_imagelayout2(
      Menu(mainmenu*,menulayout, menulayout2, imagelayout,
            imagelayout2(Layout), menuanimation) ), Layout2) = Layout

[4] distribute_layout(submenu, Layout) = distribute_layout2(submenu, Layout)
[5] distribute_layout2(image*, toggleimage) = toggleimage(image*)
```

The distribution of the image layout for main menus ((3) in figure 3.4) overlaps with the above distribution of the image layout for submenus (4) en (5). The former distribution traverses the entire `menu` source AST node and the latter only the `submenu` source AST nodes of the `menu` source AST node. Because of this dependency the first distribution must be scheduled after the latter, otherwise the images of the submenus will be layouted like the images of the main menu.

Add

Adding the non-local source AST nodes to the pivot node involves two tasks: the composition of the non-local nodes with the other nodes and the additions of the new source AST nodes to the pivot node.

In some cases the collected node(s) must be composed with the already existing child nodes of the pivot node. Composition of two already existing nodes say `A` and `B` is not directly supported by many transformation systems but only indirectly through the creation of a new node containing other already existing nodes. The only way to circumvent this is by creating a new node `C` of the same type of `A` and fill it with the other existing nodes `B` and with the nodes that were contained in the existing node `A`. This is not a problem as long as there is only one type of `A` nodes involved. As soon as there are other node types, each type must be handled separately. In the case of rewrite systems, for each node type there must be a separate rewrite rule. This increases the dependency between the grammar (defining the different node types) and the traversals. Such dependencies and their negative implications on the evolution of the grammar were discussed in section 3.4.

Recall the image layout transformation module (2) in figure 3.1 which required a set of images to apply its layout upon (see figure 3.4 for its implementation). The pivot of the rule is a set of images and the extra non-local node is the imagelayout of the menu containing this set of images. The common ancestor of the pivot and the non-local node is the `menu` AST node. The provide_imagelayout traversal (1) is triggered on the `menu` AST node. It first collects the imagelayout of the menu (2) and afterwards distributes the layout to every set of images within that same menu (3). Let us have a close look at the distribute_layout traversal, in particular at how it composes the image layout with the images. The traversal matches only when the layout (second argument) is the toggleimage layout, and thus rewrites the set of images with a new `toggleimage` node containing the images. Whenever there is a new layout, a new distribute_layout traversal must be written for it.

One could argue why the following version of the distribute_layout traversal was not used:

```
[3] distribute_layout(image*, Layout) = Layout(image*)
```

The above version no longer specifies a specific layout and should thus work for every used layout. Well, in most transformation systems the right hand side would be interpreted as a new node of type layout which contains a set of images. Naturally, this is not what was intended. Transformation systems which do support variables as the tag name of a node would interpret the right hand side as a new node which type is the node `Layout` and which contains a set of images. Again, this is not what was intended.

Now that the new source AST nodes are constructed, they must added to the pivot node. In order to add the new nodes the pivot node, the pivot node must be rewritten to another node so that it can hold the extra nodes. Recall the rearrange menu transformation module ((1) in figure 3.1) which needs direct access to its submenus. The provide_submenus traversal (1) in figure 3.2 rewrites the `menu` AST node into another `menu` AST node in which the submenus are contained.

Although this might not seem very harmful, adding these extra nodes changes the signature of the `menu` AST node. Since signatures are widely used in many transformation systems by transformation modules to distinguish between the different types of node and their subnodes, a change to the signature of anode can render a lot of transformation modules invalid, corrupt and ineliged. The title animation transformation module ((6) in figure 3.1) for example, depends on the signature of the `menu` AST node (see (2) in figure 3.3). This traversal and the menu rearrange traversals must therefore carefully be scheduled, otherwise one of the traversals conditions may suddenly fail which would in turn result in strange and hard to trace errors.

The signature dependencies of all traversals must thus carefully be analyzed and to manually order the traversals so that each traversal can safely be executed. For the our example language, here is the list of signature dependencies between the traversals for the transformation modules in figure 3.1.

- the submenu relocation before rearrange the main menu
- title animation before submenu relocation
- title animation before link resolving
- link resolving before reference resolving
- section numbering before reference resolving
- link resolving before section numbering

From the above list, the following sequence in which the traversals can safely be executed can be derived: rearrange the main menu, relocate the submenu, resolve the references, number the sections, resolve the links and animate the menu.

3.6.4 Overall process

It does not suffice to implement each need multi-source transformation module separately. The dependencies between the traversals of each multi-source transformation module must be taken into account. Without an overall process to guide the integration of their implementation, it is very likely that their traversals will interfere with the traversals of other transformation modules (cfr. section 3.6.3). These are the necessary steps of this overall process that need to be taken when integrating implementations of multi-source transformation modules:

1. **Refactor** Check whether there are transformation modules that need the same of nodes and/or perform the same computations. If this is the case, one might consider to refactor the common nodes and logic out of the transformation modules (see section 3.6.4).

2. **Order** Establish an order for the retrieval of the non-local nodes for each multi-source transformation module so that they can be found, collected, and distributed without interfering with the other traversals (see section 3.6.3).

3. **Implement** Implement the necessary traversals for each multi-source transformation module (see section 3.6.2).

4. **Clean** Group the cleaning traversals so that no nodes are removed that still might be needed by other transformation modules (see section 3.6.4).

Refactor

Some multi-source transformations require the same nodes and/or perform the same computations. This duplication is an opportunity to reduce the amount of traversals and logic in the compiler. However, such refactorings have also a negative side.

The common nodes should be refactored out of the preliminary description of at least one transformation module so both transformation modules could refer to the node that contains the common nodes. Consider for example the reference (5) and section numbering (4) transformation modules in figure 3.1 where both modules require the hierarchical number of a section. The hierarchical number of a section can first be computed and stored in the sections. This information can then be used by the references refering to these sections. Another example where refactoring can reduce the amount of logic and traversals is between the link (3) and the reference (5) transformation module in figure 3.1. Both modules require the page containing the section to which they refer in order to construct a HTML link. Instead of each computing and retrieving the page of the sections, both could rely on a separate traversal which stores the `page` node containing the section in the `section` node. Unfortunately applying refactorings is not without any risk. Refactorings introduce more scheduling dependencies which can be in conflict with already existing dependencies.

Clean

In the cookbook recipe of section 3.6.2 each multi-source transformation module is responsible for cleaning up the source AST i.e. removing the collected nodes from the original source AST nodes when those nodes were no longer necessary. The cleaning traversal was never initiated by the main traversal after the collection of the required nodes or after the distribution of the collected nodes because the decision to initiate the clean up can only be made after taking into account all the other traversals. Next, two situations are discussed which show why this is the case.

- Nodes that are needed by several transformation modules may only be removed after the traversals of the latest transformation module are executed. In our example language, this situation occurred between the title animation transformation module ((6) in figure 3.1 and figure 3.3) and the menu rearrange transformation module ((1) in figure 3.1). Both modules require the same menu animation. Since the menu rearrange transformation module transformed the whole menu, the title must be animated before the menu got rearranged and the cleanup traversal ((6) in figure 3.3) for the title animation transformation module had to be simply ignored.

- Several nodes required by different transformation modules which are contained in a same parent AST node are better removed by single traversal. Consider the image layout transformation module ((2) in figure 3.1) and the menu rearrange transformation module (1) in figure 3.1. The image layout module is scheduled before the menu rearrange module and removes the image layout from the menu ((6) in figure 3.4). Since the menu rearrange module rearranges the `menu` node and the `menu` node is changed by the image layout module, the implementation of the menu rearrange module ((1) in figure 3.1) must be adapted to cope with another kind of `menu` node. To avoid all these kind of tweaks one can combine the cleanup of the shared parent `menu` node into one traversal and schedule the traversal after the last transformation module that depends on that, in this case the menu rearrange transformation module.

3.6.5 Conclusion

Providing the non-local nodes to the pivot nodes of the multi-source transformation modules in current day transformation system is not easy. Quite lot of heuristics and guide lines need to be followed to bring the implementation to a good end and to prohibit to certain extent future evolutions. There are two major problems with the current approaches:

- The traversals often depend on each other and therefore need to be carefully scheduled. Moreover the scheduling must be done manually.
- The signature and the type of the nodes are both used as conditions to trigger transformation modules. The addition and removal of nodes therefore easily breaks the trigger conditions of other transformation modules.

```
//submenu relocation and rearranges the main menu
[1] menu(mainmenu*,menulayout(layout1),menulayout(layout2),
         imagelayout, menuanimation(layout3))
      + submenu*
            = layout3(layout1(mainmenu*), layout2(submenu*))

//imagelayout
[2] imagelayout(toggleimage)
      + image*
            = toggleimage(image*)

//link resolving
[3] link(element, ID)
      + page(ID2, ...)
      + section(ID3, ... )
            = a(element, ID2.ID3)

//section numbering
[4] section(ID1, title, ...)
      + ...
      + parent section(IDn, ...)
            = ID0.IDn...ID1 title ...

//reference resolving
[5] ref(ID)
      + ...
      + parent section(IDn, ...)
      + section(ID, title, ... )
      + page(ID0)
            = a(ID0.IDn...ID1 title, ID2.ID)

//title animation
[6] title(String)
      + page(ID, title, ..)
      + menuanimation(animation)
      + Link(element, ID)
            = animation(title(String))

[7] section(ID, ... )
      + Link(element, ID)
            = a(ID1, section(ID, ...))
```

Figure 3.1: The transformation modules that need several indirectly connected source AST nodes.

```
[1] provide_submenus(
      menu(mainmenu*,menulayout,menulayout, imagelayout, menuanimation))
      = menu(mainmenu*, collect_submenus(mainmenu*, ),
             menulayout, imagelayout, menuanimation)

[2] collect_submenus(sub, sub*) = sub sub*

[3] cleanup(main(ID,element, submenu*)) = main(ID,element)
```

Figure 3.2: The traversals for submenu resolving

```
syntax-functions
      pageanimations(ID*, Animation) -> menulinklayout

equations
      [1] provide_titleanimation(Root)
            = distribute_titleanimation( collect_titleanimation(Root, ) )

      [2] collect_titleanimation(
            Menu(mainmenu*,menulayout, menulayout2,imagelayout, menuanimation), ID*)
            = pageanimations( collect_pages(mainmenu*, ID*)
                              + collect_pages(submenu*, ID*),
                              collect_menuanimation(menuanimation, ))

      [3] collect_menuanimation(menuanimation(Animation1), Animation) = Animation1

      [4] collect_pages(Link(Element,ID), ID*) = ID ID*

      [5] distribute_titleanimation(Page(ID,title,element*),
                           pageanimations( ID1* ID ID2*, Animation(slide)) )
               = Page(ID, i_slide(Title), element*)

      [6] cleanup( Menu(mainmenu*,menulayout, menulayout2,imagelayout, menuanimation) )
            = Menu(mainmenu*,menulayout, menulayout2,imagelayout)
```

Figure 3.3: The traversals for title animation

```
[1] provide_imagelayout(menu)
      = distribute_imagelayout(menu, collect_imagelayout(menu, ))

[2] collect_imagelayout(imagelayout(Layout), Layout2) = Layout

[3] distribute_layout(image*, toggleimage) = toggleimage(image*)

[4] cleanup(Menu(mainmenu*,menulayout, menulayout2, imagelayout,
                 imagelayout2(Layout), menuanimation) )
      = Menu(mainmenu*,menulayout, menulayout2, menuanimation)
```

Figure 3.4: The traversals for image layout

Chapter 4

Generator Composition Problems

In this chapter, we deal with the composability of program generators. This is required to enhance reusability of program generators across different development contexts.

4.1 Introduction

A traditional program generator is generally considered in isolation. Contemporary program generators produce program parts that are encapsulated modules with a well-defined interface. The internals of the generated modules are completely determined by a single program generator, i.e. the generated modules are *black-box*. The kind of modules depends on the target language in which the program part is generated. For example, in a class-based object-oriented language, the generated program parts are classes on which a set of methods are defined. As a consequence, the use of multiple such program generators in the development of an application entails that the developer needs to write glue code to make the generated modules work together. In such a setting, the program generator composition problem is quite equivalent to the component composition problem , i.e.: how do we correctly compose black-box program modules? The difference with classic components is, of course, the use of generative technology such that the component itself can be customized, providing more flexibility compared to library components.

However, many frequently implemented program parts *are* not or *can* not be implemented using encapsulated modules. Such program parts are actually collaborations between multiple modules (e.g. publish-subscribe dependencies) or conceptual modules that are spread across different implementation modules (e.g. synchronisation). The integration of such non-encapsulated program parts in modules produced by other generators cannot be achieved by hand without breaking the black-box property of the generated program parts. The manual integration requires detailed knowledge on the internals of the modules and might result in broken functionality.

While writing low-level glue code to manually compose traditional reusable components is a completely natural scenario, it is undesirable in the case of generated components and even problematic or almost impossible when these generated parts are not even encapsulated modules. There are several reasons for this. First, the interface (and behaviour) of a generated part might change whenever the specification of the generated part is adapted. Since this specification is part of the program itself, the specification is susceptible to changes at any time. Second, the composition of the generated parts cannot always be expressed in terms of the public interfaces of the generated parts. Such a composition is either impossible to achieve or it would force a developer to break the black-box property of the generated part. If this latter would be possible, the developer would still have a hard time understanding the generated code and making the correct adaptations to it. Third, the integration of non-encapsulated program parts in generated encapsulated parts might break the functionality of both parts quite easily. This can be caused either by low-level (syntactic) conflicts or more high-level (semantic) problems. Last but not least, the integration of non-encapsulated parts requires that the developer is confronted with the low-level details of the generated program parts.

As such, compositions of generated program parts are not practical and can often result in incorrect compositions. Therefore, we propose to compose the program generators instead of the generated parts. A composed generator then produces the composition of the generated parts. In the following section, we briefly explain why compositions of generated components are undesirable.

4.2 Composition of Generated Parts

The most straightforward approach to use several program generators in the development of a software application is generating the required parts separately and composing these parts by hand. This means that the generated parts can be composed by the developer in the same way as standard reusable components from a library can be composed. This approach is quite feasible if the generated parts are encapsulated modules that are independent of any other generated parts.

Everything becomes quite different when the generated program parts are not independent of other generated parts or when the generated parts are not even encapsulated modules. In those cases, developers cannot threat the generated parts as black-box components. Once the generators have produced their output code, the developer will need to write program code or even change generated code such that the generated program parts communicate and interoperate. In these cases, the use of program generators only alleviates much of the initial coding effort for the separate program parts. A lot of manual work still needs to be done in verifying and correcting the generated parts. Moreover, future evolutions have to repeat this work or may also introduce new problems. In practice, this also means that the generators need to be executed in a certain order. This order may or may not be constrained by the generators themselves, based on their input requirements and produced output code.

For example, for the development of a new compiler, developers will often use a parser generator. The parser generator accepts a grammar specification and will generate both a parser and a parsetree structure. The development environment also offers them the possiblity to generate frequently-used design patterns, such as the visitor design pattern. They plan to use the visitor design pattern to implement traversals of the parsetree for compilation, pretty-printing, optimizations, etc.... To do this, they need to manually specify what generated classes are actually parsetree classes and they also need to know how a parsetree will look like at runtime. This requires them to have some knowledge on the generated code. At that moment, the developers doe not consider this a problem since they need to know what the parsetree looks like for other parts of the program as well. As such, they do not assume this to be a disadvantage. However, once they try to compile the program, the program does not work at all and strange errors appear. After investigation, they notice that the visitor generator generates an accept method on each of the parsetree classes, thereby overwriting the original accept method generated by the parsetree generator. The parsetree already contained an accept method that is used by the parser to build the parsetree. Therefore, they have powerful refactoring tools and decide to modify the implementation of the parser and parsetree such that the accept method, generated by the parsetree generator, is renamed. Afterwards, they regenerate the visitor and everything works as expected. Some months later, a change to the language specification is made, resulting in a change of the grammar specification. They regenerate the grammar and suddenly the compiler does not work anymore. After some time, somebody recalls that they had to modify the generated code such that it would not conflict with the visitor generated code. They apply this refactoring but the visitor generator is unable to regenerate because it cannot find the right parsetree classes. At that moment, they also realize that the input specification of the visitor generator was based on the generated code for the old grammar specification.

Depending on the kind of generated parts, the composition of the parts with the rest of the program requires little or deep understanding of the generated code. Clearly, the intention of program generation is not only to generate code for frequently used structures, but also to hide its implementation details. As such, the more a developer needs to know about the generated code to make the software work, the less advantage the use of program generation has brought to the development of the application. Moreover, as the interoperability code for each generated part is manually written, the implicit dependencies between the generated parts will often be broken each time a generated part is regenerated. Since these implicit dependencies need to be manually revised after each regeneration, the more implicit dependencies between generated parts, the less advantage in using program generators.

In a best-case scenario, the generated program parts are encapsulated components that communicate with the rest of the program through a well-defined interface. In order to correctly compose the generated program parts, the developer only requires knowledge about the interface of the generated part. In this case, the composition is technically identical to the composition of traditional reusable components. The most important disadvantage here is that extra glue code and adapters need to be written, introducing extra technical code and indirections in the program. More importantly, the manual written code introduced implicit dependencies between the generated parts.

In a worst-case scenario, one or both generated parts crosscut other components in the program. Their mutual composition, or even the composition with manually written code often requires manual adaptations to the generated parts. Furthermore, the developer needs a deep understanding of the generated code and the composition becomes hard and might brake generated functionality. This scenario often results in the developers needing to understand the entire generated part, resulting in a practical disadvantage of using a generator at all.

Bibliography

[1] Aho, Sethi, and Ullman. *Compilers: Principles Techniques and Tools.* Addison-Wesley, 1986.

[2] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. In *Proceedings of the 5th Int. Conf. on SoftwareReuse.* IEEE, 1998.

[3] D. Batory, C. Johnson, B. MacDonald, and D. 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.

[4] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *Automated Software Engineering Conference*, pages 81–92, 2002.

[5] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. *ACM SIGSOFT 93: Symposium on the Foundations of Software Engineering*, December 1993.

[6] 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.

[7] J. Brant and D. Roberts. Smalltalk compiler compiler.

[8] S. Chiba. A metaobject protocol for c++. In *Proceedings of the OOPSLA Conference*, pages 285–299, 1995.

[9] K. Czarnecki. Gcse working group.

[10] K. 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.

[11] V. S. Don Batory, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. Achieving reuse with software system generators. *IEEE Software*, September 1994.

[12] FOLDOC. Free online dictionory of computing.

[13] A. Goldberg and D. Robson. *Smalltalk-80: the language.* Addison-Wesley, 1983.

[14] O. M. Group. Model-driven architecture.

[15] W. Harrison and H. Ossher. Subject-oriented programming - a critique of pure objects. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications.* ACM, 1993.

[16] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly.

[17] P. K. M.G.J. van den Brand. *ASF+SDF Meta-Environment User Manual.* Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, release 1, revision 1.134 edition, September 2003.

[18] Microsoft. Using the codedom, in the .net framework developer's guide.

[19] O. D. Moor, G. Sittampalam, and E. van Wyk. Intentional programming: a host of language features. Technical report, Oxford University Computing Laboratory, 2001.

[20] J. Neighbors. *Software Construction using Components.* PhD thesis, University of California, Irvine, 1980.

[21] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. *Specifying Subject-Oriented Composition*, volume 2. Wiley & Sons, 1996.

[22] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical report, IBM T.J. Watson research center, apr 1999.

[23] F. Rivard. Smalltalk: a reflective language. In *Reflection'96*, 1996.

[24] Y. Smaragdakis and D. Batory. DiSTiL: A transformation library for data structures. In *Domain-Specific Languages (DSL) Conference*, pages 257–270, 1997.

[25] Y. Smaragdakis and D. 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.

[26] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[27] Y. V. Srinivas and R. Jullig. Specware: Formal support for composing software. In *Mathematics of Program Construction*, pages 399–422, 1995.

[28] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. Openjava: A class-based macro system for java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[29] D. Tidwell. *XSLT*. O'Reilly, 2001.

[30] M. G. J. Van Den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[31] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[32] J. van Wijngaarden and E. 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.

[33] M. Voelter. Program generation: a survey of techniques and tools.