An Illumination of the Template Enigma

Software Code Generation with Templates

B.J. Arnoldus

An Illumination of the Template Enigma

Software Code Generation with Templates

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 1 februari 2011 om 16.00 uur

door

Bastiaan Jeroen Arnoldus

geboren te Amstelveen

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.G.J. van den Brand

Copromotoren:
dr.ir. J.J. Brunekreef
en
dr. A. Serebrenik

An Illumination of the Template Enigma

Software Code Generation with Templates

B.J. Arnoldus

Promotor:	prof.dr. M.G.J. van den Brand (Technische Universiteit Eindhoven)
Copromotoren:	dr.ir. J.J. Brunekreef (Hogeschool van Amsterdam) dr. A. Serebrenik (Technische Universiteit Eindhoven)

Overige leden kerncommissie:
prof.dr.rer.nat.habil. U. Aßmann (Technische Universität Dresden)
prof.dr. P. Klint (Centrum voor Wiskunde en Informatica)
prof.dr. J.J. Lukkien (Technische Universiteit Eindhoven)

Het werk in dit proefschrift is uitgevoerd onder auspiciën van de onderzoekschool IPA (Instituut voor Programmatuurkunde en Algoritmiek).

IPA dissertation series 2011-02.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-2418-1

Druk: Printservice Technische Universiteit Eindhoven
Omslagontwerp: Raymond van der Aa

Voor:
Antonius Arnoldus
Monique Arnoldus-Zuiver
en
Martijn Jan-Willem Arnoldus

Dankwoord

Kort wil ik de mensen bedanken die me hebben geholpen bij mijn onderzoek en het schrijven van mijn proefschrift. Als eerste mijn promotor Mark van den Brand, die mij tijdens een college verzocht "na te blijven". Ik wil je bedanken voor de discussies, het delen van gedachten en de begeleiding om dit proefschrift gestalte te geven. Naast de promotor werd ik bijgestaan door de copromotoren Jacob Brunekreef en Alexander Serebrenik. Jacob dank ik voor voor de begeleiding vanuit de Hogeschool van Amsterdam, waar je ervoor zorgde dat ik mijn onderwerp kon behouden. Alexander ben ik erkentelijk voor de ondersteuning bij het formaliseren van mijn theorieën en gedachten. Jeanot Bijpost dank ik voor zijn begeleiding vanuit Mattic.

Ik wil de overige leden van de leescommissie, bestaande uit prof.dr.rer.nat.habil. U. Aßmann, prof.dr. P. Klint en prof.dr. J.J. Lukkien, bedanken voor het beoordelen van mijn proefschrift. Tijdens de verdediging zal ik worden begeleid door de paranimfen Marcel van Amstel en Stephan de Gruijter: sine protectione nulla securitas.

De studenten die, gedurende mijn onderzoek, projecten hebben uitgevoerd: Robin Boon, Wouter Bruggeman, Arjan van der Meer, Jos Peeters, Guido Smeets, Maurice van der Star en Aram Verstegen. Ik dank medepromovendi en Braga '06 lotgenoten: Superbockmeister Zvezdan Protic en Luc –is lost– Engelen. Loek Cleophas: nooit gedacht dat ik ooit je proefschrift "stuk" zou lezen. Ik dank Martin Bravenboer voor de discussie op het gebied van ambiguïteiten, Jurgen Vinju voor de technische ondersteuning bij ASF+SDF en Sander Bakkes voor zijn LaTeX bestanden.

Ik ben de Hogeschool van Amsterdam en namens de hogeschool, Marjan Freriks en Kees Rijsenbrij, zeer erkentelijk voor het mogelijk maken van mijn promotieplek. Ik bedank Wilko Oskam voor het vinden van de parel in het woud der woorden en Raymond van der Aa voor de uitbeelding. Zonder Arnim Eijkhoudt en Jan Derriks was er geen bereikbare Oege. Hayco de Jong zorgde voor een segmentation fault. Zonder Yvette Letiche-le Noble, Janet Hofstra en Mark Thomas had ik geen toegang tot Wandelgangen Informatiesysteem. Verder dank ik mijn oud-kamergenoten Anneke Baas, Daan van den Berg, Wilma van Hoogenhuyze en Hans van Koolbergen.

Tot slot, Niels van Galen Last, Robin den Held, Dominiek ter Heide, Tristan Suerink, Yme van der Velden, Sil Westerveld; stealth start-ups, "Bestijg de trein nooit zonder uw valies met dromen", 15-daagse ensembleverwachtingen, bandbreedte, ambulances, Paul, Kanji, maar vooral ntsk ntsk ntsk.

Jeroen Arnoldus
amsterdam, december 2010

Contents

Cover **i**

Contents **xi**

1 Introduction **1**

1.1 Process of Designing 1

1.2 Level of Abstraction 3

1.3 Code Generation 6

1.4 Homogeneous Code Generators 9

1.5 Heterogeneous Code Generators 12

1.6 Conclusions 23

1.7 Problem Statement 24

2 Preliminaries **27**

2.1 Basic Definitions and Notations 27

2.2 Context-free Grammars 31

2.3 Regular Tree Grammars 33

2.4 Relations between CFL and RTL 34

2.5 Abstract Syntax Trees 36

2.6 Used Languages and Formalisms 37

3 The Unparser **43**

3.1 Introduction 43

3.2 Deriving Abstract Syntax Trees 44

3.3 The Unparser 50

3.4 Unparser Completeness 54

3.5 Conclusions 58

4 The Metalanguage **61**

4.1 Introduction 61

4.2 Code Generators 62

4.3 Our Metalanguage 65

4.4 Example: The PICO Unparser 86
4.5 Related Template Systems 86
4.6 Conclusions 101

5 Syntax Safe Templates 103
5.1 Introduction 103
5.2 Syntax Safe Templates 104
5.3 The Metalanguage Grammar 110
5.4 Grammar Merging 116
5.5 Related Work 118
5.6 Conclusions 119

6 Repleo: Syntax Safe Template Evaluation 121
6.1 Introduction 121
6.2 Syntax Safe Evaluation 122
6.3 Substitution Placeholder 123
6.4 Match-replace Placeholder 124
6.5 Subtemplate Placeholder 125
6.6 Substitution Placeholder Revisited 126
6.7 Ambiguity Handling 128
6.8 Separator Handling 131
6.9 Repleo 132
6.10 Case Studies 132
6.11 Related Work 142
6.12 Conclusions 144

7 Case Studies 145
7.1 Introduction 145
7.2 Code Generator Architectures 146
7.3 Metrics 150
7.4 A DSL for Web Information Systems 151
7.5 ApiGen 157
7.6 NunniFSMGen 173
7.7 Dynamic XHTML generation 189
7.8 Conclusions 200

8 Towards Static Semantic Validation 203

8.1 Introduction 203

8.2 Static Semantics 204

8.3 Defining Static Semantic Checkers 205

8.4 Connecting SGLR and JastAdd 210

8.5 Static Semantic Checks for Templates 211

8.6 Related Work 218

8.7 Future Work 219

8.8 Conclusions 221

9 Conclusions 223

9.1 Contributions 223

9.2 Future Work 228

9.3 Final Conclusions 229

Summary 243

Samenvatting 247

Curriculum Vitae 251

1 Introduction

Creating software is a process of refining a concept to an implementation. This process contains several stages visualized by documents, models and plans at several levels of abstraction. Mostly, the refinement process requires creativity of the programmers, but sometimes it is a boring repetitive task.

This repetitive work is an indication that the program is not written at the most suitable level of abstraction. The level of abstraction offered by the used programming language is probably too low to remove the recurring code. We focus on using code generators to raise the level of abstraction and to automate the repetitive work. In this chapter we present a number of approaches to implement a code generator.

We discuss the advantages and disadvantages of the different approaches. This thesis will focus on code generators using templates. We present the research questions in the context of raising the quality of template based code generators.

1.1 Process of Designing

Designing is the process of deciding how something will look, work, etcetera, by drawing plans, making models, and so on [2]. One of the design methods is to stepwise refine an abstract idea or concept to a full-fledged, producible design. This way of designing, or developing, is used in a broad range of disciplines; from storyboard to movie, from equation to electrical amplifier, from sketch to painting and from software requirements specification to computer application. This design method or development method can consist of several intermediate plans, designs or models, which are created in a linear or iterative process.

For example, Figure 1.1 shows an unfinished painting of Rubens. The painting provides a showcase of the techniques and methods used by Rubens. Rubens used an underpainting, still visible in the unfinished painting, as an initial layer. It helps to define color values for the later painting and to define the

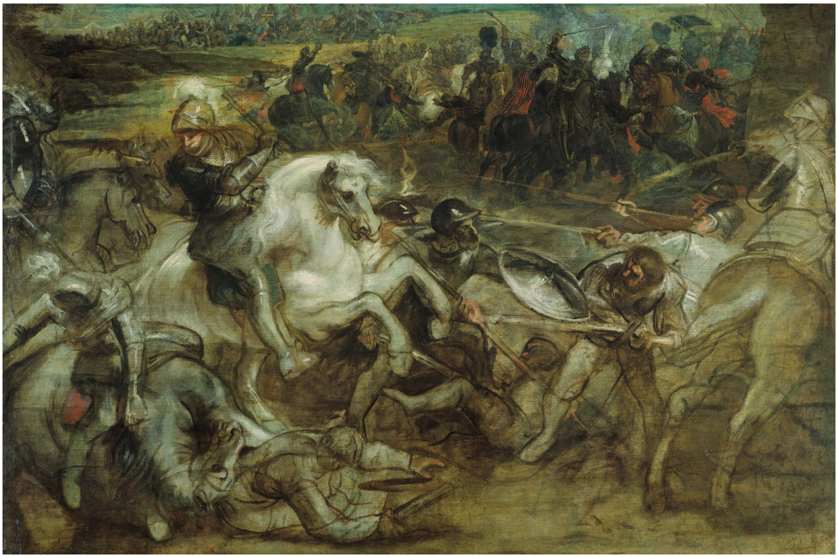


Figure 1.1 Unfinished painting of Rubens (Henry IV at the Battle of Ivry, Peter Paul Rubens 1628-1630) [101].

shapes of the final figures. The underpainting can be seen as a kind of *model* of the final painting. In general, a model is a particular design of a product, or it is a simple description of a system, used for explaining how the system works [2]. We consider the underpainting as a model for the final painting as it is a design of the final result.

In case of this work of Rubens, the underpainting shows a kind of iterative process to find the right shapes and composition[1]. The painting consists mainly of long sketch lines, where some places are more finished and thus show more detail, like the head of the horse in the middle of the painting. Consider the soldier in the middle right behind the horse in the unfinished painting; the soldier has three arms, one right arm with a sword, one right arm with a lance and a left arm with a shield. This indicates that Rubens was looking, in a kind of iterative "trial and error" process, to find the right shape for this soldier, i.e. with a sword or a lance.

One can observe the same kind of process as used by Rubens during the development of software. Requirements are collected, based on the requirements models defining behavior and data structures are developed, and finally these models are refined and merged until a working application is created. Many of these refinement steps in this process require intelligence and creativity of

[1] http://www.rubensonline.be/showDetail.asp?artworkID=100575 (accessed on November 30, 2010)

the programmers, but sometimes it is necessary to perform boring repeating uncreative work. One of the challenges in computer science is to automate the uncreative work by means of raising the *level of abstraction* of the used programming languages.

1.2 Level of Abstraction

Floridi et al. provide a definition of *level of abstraction* [43], wherein each level represents a finite but non-empty set of *observables*. An observable is a typed variable, not necessarily meant to result from quantitative measurement. A *typed variable* is a uniquely-named conceptual entity (the variable) and a set, called its type, consisting of all the values that the entity may take. For example, in case of describing a wine, one can define the observable *color*, which has a value from the set $\{white, golden, red, ruby\text{-}red, \ldots\}$. Observables belong to a certain level of abstraction. An observable (or a set of observables) at a higher level of abstraction can be refined to a set of lower level observables, where this set of observables contains more elements than the set of observables at a higher level of abstraction. In other words, when considering a representation at a lower level of abstraction, the used observables are more granular and relatively more concrete. For example, in case of wine the level of abstraction containing the typed variable *tasting* can be refined by the typed variables *nose*, *robe*, *color*, *acidity*, *fruit* and *length*, which represent a lower level of abstraction. Another example of level of abstraction contains the typed variable *purchasing*, which is refined by the typed variables *maker*, *region*, *vintage*, *supplier*, *quantity* and *price*.

The examples for wine abstractions also show that it is usual to have different levels of abstraction describing different views of a concept. The *tasting* value is more relevant while evaluating a wine and the *purchasing* value more relevant for ordering a wine. The use of different models makes it easier to communicate about the concept with different people with different backgrounds. In the discipline of Software Architecture these models are called *views* of the system used to discuss with the different stakeholders [11]. Beside the aspect of communication, models enable analysis, verification and optimization at the most suitable level of abstraction. This would have been much harder, if not unfeasible, when the models are merged and/or refined to a model at a lower level of abstraction, since it would be too complex to recognize the higher level artifacts in the lower level representation [56].

Code expressed in a programming language is a model of behavior, which results from executing the code. We consider a programming language to be more abstract, when it is necessary for execution to expand its instructions to more fine grained instructions in a language of a lower level of abstraction. This closely relates to the definition of level of abstraction given by Floridi. To illustrate, one can identify the following levels of abstraction in a personal computer: electronic circuits, gates, microprocessor and memory, instructions in assembler, programming languages mapped on assembler, operating system described in programming languages and applications described in programming languages unified with the operating system. Each level of abstraction hides implementation details used in lower level of abstractions. For example logic gates can be implemented using transistors, relays or tubes as long as the implementation fulfills the requirements of the logic gate. Another example is a program written in C, not using operating system calls, can be compiled for an Intel X86 architecture, Sun Sparc architecture or embedded Atmel AVR architecture.

A language with a higher level of abstraction does not need to have knowledge of the lower level implementation. The implementation knowledge is embedded in the translation function from the high level code to a lower level implementation. However, in practice one can notice influences of the lower levels of abstraction on the higher levels of abstraction, sometimes called *leaky abstractions* [107]. For example, the performance of iterating over a large two-dimensional array depends on the memory organization, whether the iteration could be better implemented horizontally rather than vertically. Another example is the way an SQL statement is interpreted. The statements

```
where a=b and b=c and a=c
```

and

```
where a=b and b=c
```

result in the same set of records, but depending on the database implementation the first one could be faster. Besides performance issues, leak of abstractions lies at the heart of a number of portability and complexity problems [67]. An example of portability problems caused by leak of abstractions is the use of virtual memory. In order to improve the performance, objects must be allocated within the same memory page. When the code is compiled for a new platform, the apparently portable code could be extremely slow or even crash.

In this thesis we focus on programming at a higher level of abstraction and the translation of code to a lower level implementation. Programming using a language with a higher level of abstraction, offers a language with a syntax that is better suited for the application area, since it hides low level details not particular related to the application area. Automating programming is only possible when the more abstract language can be translated in one or more steps to a language that is executable by a computer. This requirement especially holds for systematic model refinements, consisting of repetitive non creative work. We study the implementation of programs writing other programs, i.e. *code generators*, in this setting of raising the level of abstraction. Code generation is a technique that is mainly, but not exclusively, used to translate code written in a programming language at a high level of abstraction to code written in a less abstract target programming language.

In the domain of computer programming it is already common practice to use *general-purpose* languages at a high level of abstraction with respect to the machine. A general-purpose programming language, such as Java, is designed for a broad spectrum of *application domains*. An application domain is a set of related systems that share design characteristics. Compilers are used to translate code written in these programming languages into machine code instead of programming directly in machine language. However, these higher level programming languages are still machine oriented. They do provide abstraction for register allocation, memory management and so on, but not for a particular application area, like information systems, industrial processes or even GUI's. Although (external) libraries and frameworks can amplify the feeling, that a language is geared towards some application domain.

The broad applicability of general-purpose languages results in a lack of *expressiveness* in a particular application domain. The expressiveness, here meant, is the level of compactness to state something. There is a trade-off between expressiveness in a particular domain of a language and the general-purpose character of the language. This trade-off can be informally defined as

$$\textit{expressiveness} \times \textit{domain size} = \textit{constant}\ [56].$$

A language better suited for a particular application domain has a higher level of abstraction than a general-purpose language, since concepts of the less generic language can be translated to a more verbose implementation in the general-purpose language. At a certain point the domain of a language is so limited that the language is called a domain specific language (DSL) [80]. The point where a language becomes domain specific is not quantified. For some people Cobol is a DSL, while other people consider languages like HTML or SQL as a DSL. Being acquainted with the level of abstraction of the language is indispensable for knowing the scope and limits of the language [43]. A more abstract language has a greater distance from the machine, although some languages provide *escape* clauses to define syntax in a lower level of abstraction, like inline assembler in C.

A general-purpose language with the ability to specify libraries can act as a domain specific language [56]. The domain specific vocabulary is collected in functions, classes and parameterized types (C++ Templates or Java Generics). These extensions can be considered as raising the level of abstraction within the programming language, given that after the definition of functions and classes "new syntax", in the form of types and function calls, can be used. During compile time or run-time this "new syntax" is internally expanded to an implementation. One of the implicit tasks of a programmer is to continuously implement domain specific extensions expressed in the syntax of the used programming language. During programming a programmer specifies functions and classes to organize and reuse code. Sometimes a set of functions and classes reaches such a high level of re-usability for a particular domain that they are collected in a library.

The possibility of extending a programming language with vocabulary for new behavior and types, and thus raising the level of abstraction, is limited by the support of syntax to write these extensions. There exist languages with many options to write these extensions, like MetaML [110] or Python[2]. MetaML allows evaluating functions partially in order to get specialized instantiations. In Python everything is an object, even the code. Since the code is an object, it is possible to manipulate the object of the code in order to *change* it on-the-fly. Many languages, such as C and C++, lack these features. An example of this limitation is shown in [10]. The authors have shown that the C++ Standard Template Library still contains a lot of code clones, despite of the heavily use of type parameterization offered by C++ Templates. Another example of a limitation of the programming language is the declaration of exception handling in Java. The thrown exceptions in Java cannot be parameterized and code clones are necessary when code is used which must throw another exception in another context [69]. Code cloning can become a serious issue when a language cannot express these variations. Source code containing very large code clones is less reliable than code without clones and source code containing clones is less maintainable [81]. Code generation can be a solution to raise the level of abstraction and encapsulate these variations. The next section discusses the properties of a code generator and a couple of frequently used implementation approaches.

1.3 Code Generation

A *code generator* is a program generating other programs. It is a subclass of *meta-programs*; the set of programs analyzing or manipulating other programs [103]. In this thesis we limit ourselves to *static* code generators. In this class of code generators the generated code is written to disk, or another representation of a file, before it is processed in a next stage. The opposite case is *run-time* code generation, where the generated code is executed immediately. In this thesis, internally generated code, such as inline function expansion or template instantiation in a compiler, is considered as run-time code generation. Otherwise, language features such as expanding inline function calls can also be seen as a form of code generation.

Code generation is a projection of some *input data* to output code. This input data conforms to a language with its own syntax and semantics, independently

[2] http://www.python.org (accessed on November 30, 2010)

defined of the code generator. A code generator translates the input data into another representation, often a representation at a lower level of abstraction. Therefore it manipulates data, representing sentences in a language, i.e. the output code. This output code can be everything, from machine code, in case of compilers, to code of a general purpose programming language, in case of computer-aided software engineering or model-driven engineering tools [102, 42].

Inside a code generator code artifacts play two roles. The first role is code acting as data, this code is called the *object code*, and is used as building blocks to construct the *output code*. The second role is the code manipulating the object code during generation, this code is called *meta code*. The meta code and the object code can be expressed in the same language, this is called a homogeneous system, or the meta code and object code can be expressed in different languages in a heterogeneous system [103].

Some disambiguation should be made with respect to the use of the word *meta* in this thesis. Consulting the dictionary [2] *meta* is used as a prefix and based on the context it means "connected with change" or "beyond". The meta code is expressed in the so called *metalanguage*. The dictionary defines metalanguage as the words and phrases that people use to talk about or describe language in general or a particular language. In the discipline of linguistics, metalanguages, such as Natural Semantic Metalanguage [47], are developed to express statements about other natural languages, called object languages, in semantic primitives. The word meta is used in a different way when discussing code generators. In the context of code generators, terms with the prefix *meta* denote the artifacts *manipulating* the object code instead of describing the object language. We define the metalanguage as the language in which the code generator is written. The object language is the language of the code that is manipulated.

There are multiple reasons to choose for code generation [120]. The main reasons for using code generation can be categorized in three topics: using an object language independent of the metalanguage, raising the level of abstraction, or improving the performance during execution. Using an object language independent of the metalanguage is a requirement when it is impossible to use a metalanguage identical to the object language, i.e. homogeneous code generators. One can think of compilers translating third generation languages to assembler, or, maybe more important, code generation for communication reasons, like HTML. Object languages without the possibility to express operations, such as HTML, cannot act as metalanguage and thus homogeneous code generators are no option for these languages.

The second reason for using code generation is raising the level of abstraction. Code generation offers a mechanism to program at a higher level of abstraction, when the used programming language does not offer syntax to express it. For example a language does not offer the notion of classes natively.

The last reason is performance. In case the language offers syntax to define generic constructs, the use of it may at an execution performance penalty. Code generation can be used to improve the execution performance of an application, without losing the level of abstraction in the implementation. During code generation it can pre-calculate the specialization of the code, so that the overall performance is better during run-time. Calculations are already performed during generation time. Performance is sometimes better predictable, for instance when recursive functions are pre-evaluated. Prediction of time and memory usage is especially important in embedded systems, where resources are limited.

One of the aims of software engineering is to reuse as much code as possible. Just as the applicability of domain specific libraries, the availability of off-the-shelf code generators is limited. Since code generation is used for raising the level of abstraction, problems solved by a code generator are often very application specific or domain specific. If a code generator seems to exist for a specific task there are a few barriers to reuse it. In short, these barriers are; the output language differs from the used programming language, the code is not readable and thus not reviewable, and the interfaces do not fit in the application architecture [98]. These problems can be removed by writing a proprietary code generator tailored for its task, like proposed in [58].

Unfortunately, constructing code generators is not a trivial task. First, the code generator should offer the best suited patterns for its level of abstraction and problem domain. Finding these patterns requires an extensive knowledge of the problem domain. This requirement is not limited to writing code generators, but also for writing understandable and reusable domain specific libraries. Second, a code generator contains code artifacts executed at different stages. A code generator contains code executed at generation time, *meta code*, and code as data used in building blocks for the output code, the *object code*. A developer has to be aware continuously of the different execution stages of the different code artifacts. More inconvenient, artifacts belonging to an execution stage are mixed with artifacts belonging to the other execution stage(s).

Finding errors in code generators without tool support is hard. Errors in the meta code are detected during compilation or during the execution of the code generator. Errors in the output code are harder to find and are detected when the output code is compiled or executed. In many code generators the object code is still represented as strings without any internal structure [103]. Debugging these errors is time consuming, since the code generator has to be corrected, it has to be compiled, code has to be generated and finally the generated code must be tested. Debugging tools for the meta code do not help, because the object code is not executed at code generation time and is seen as data, not as a program.

There are code generator implementation approaches providing safety. We define three classes of safety a code generator can provide:

1. no safety;
2. syntax safety;
3. type safety.

The first class of safety contains the code generators providing no guarantees of the correctness of the output at all. Code generators without notion of the structure of the output code handle the output code as a sequence of characters. Beside the consequences for the testing and debugging of this class of code generators, it can even lead to serious security flaws. One of the top 25 security bugs caused by these code generators is HTML injection in web applications[3], a vulnerability allowing an attacker to inject browser-executable content into a dynamic generated web page. The code generator cannot distinguish good code from malicious code since it only considers the output as a sequence of characters. Most code generators belong to this class.

The second class of safety contains the code generators guaranteeing that the output code is at least syntactical correct. A code generator is syntax safe when for every input it is guaranteed that the generated code is a valid sentence of the output language.

The third class of safety contains the code generators guaranteeing that the generated code is also static semantically correct. Static semantics are requirements for the code, which cannot be detected by only parsing it, such as checks for double declared variables and type errors.

The next sections will show a number of code generator implementation approaches and the safety level they provide.

1.4 Homogeneous Code Generators

As mentioned earlier there are two distinct kinds of code generators: homogeneous and heterogeneous [103]. Homogeneous is the collective term for all meta-programming approaches with equal metalanguage and object language. These systems have numerous advantages above heterogeneous systems [103]:

- ◇ homogeneous systems can be multi-level, where an object-program can itself be a meta-program that manipulates second-level object-programs.

[3] http://cwe.mitre.org/top25/ (accessed on March 11, 2010)

```
fun power n = (fn x => if n=0
        then < 1 >
        else < ~z * ~(power(n−1) x) >)

map(run < fn z => ~(power 3 < z >) >) [2,4]
```

Figure 1.2 MetaML example.

- In a homogeneous meta-system a single type system can type both the meta-language and the object-language.
- Homogeneous meta-systems can support reflection, where there is an operator, "run" or "eval", which translates representations of programs. This is what makes run-time code generation possible.

A typical example of a homogeneous meta-programming system allowing *staged programming* is MetaML [110]. ML is a general-purpose functional programming language. MetaML is a homogeneous meta-programming approach using ML. It makes it possible to generate ML inside a ML program in the context of staged programming. Staged programming is a technique introducing the possibility to control the order of evaluation, sometimes called partial evaluation. This technique enables the use of a very generic function, specialize it at compile time and use the specialized version at run-time. Specifying a precise execution order allows the programmer to control program resources, like time and space. Whether a statement or term is executed during run-time or compile time is not a choice of the compiler, but an explicit decision of the programmer. For example a recursive function can be unfolded during compile time, so that the run-time memory usage and execution time can be predicted.

Figure 1.2 shows a staged power function defined in MetaML. The result of the `map` is `[8,64]`. In normal execution without staging, for each value in the list the power function is executed. Since the power function is only used as the cubic function, in a non staged execution every time the same recursion is executed. By separating this process in two stages, the calculation of the cubic function can be done once instead of every time the power function is called. As shown in the example, first the recursion is calculated via the `run` instruction, resulting in the cubic function `map( fn z => z * z * z * 1 ) [2,4]`, and in the second stage the map function is evaluated.

It should be noted that MetaML programs are complete compilation units and that the program is executable without staging annotations. This allows the possibility of strong typing and guarantees the syntactical correctness of the expanded code. In fact the parser of ML can be used, with added grammar rules for the stage annotations and the ML type checker with extra scope rules and bind rules for the variables in the different stages.

```
#include <iostream>

template <typename T>
T GetMin(T x, T y)
{
   if(x < y) return x;
   return y;
}

int main () {
  int a=2, b=7;
  long c=20, d=4;
  std::cout << GetMin<int>(a,b) << std::endl;
  std::cout << GetMin<long>(c,d) << std::endl;
  return 0;
}
```

Figure 1.3 C++ Template example.

We make the definition of a homogeneous system more liberal, by defining a homogeneous system as one where all the components are specifically designed to work with each other, whereas in heterogeneous systems at least one of the components is largely, or completely, ignorant of the existence of the other parts of the system [111]. Using this new definition we can consider the following systems also as homogeneous systems: C++ Templates [112], Template Haskell [104] and Java Generics [17]. As these homogeneous systems have a lot in common, we will restrict ourselves to discuss C++ Templates.

C++ Templates are a language feature that allows to define functions and classes based on generic types. Functions or classes can work on many different data types without being rewritten. This is especially useful, when functionality depends on the underlying structure and not on the kind of data to operate on. Examples are stacks, lists, queues and sorting algorithms. As an unintended feature, C++ Templates also support staged programming [113].

An example of type parameterization via C++ templates is shown in Figure 1.3. This example shows the definition of a generic comparison template `GetMin`. This template is invoked in the body of the `main` function, where it is parameterized with the desired type via the <`type`> syntax. The same function can now be used for any numeric type.

Although homogeneous code generators have certain advantages over heterogeneous code generator approaches, it is not always possible to use them. The benefits only hold when code for the same language must be generated, then is full support for syntax checking and type checking offered by the compiler. In a heterogeneous situation, where code for another language must be generated, the internal code expander of the compiler cannot be used. As a result the safety offered in the homogeneous situation is lost.

The discussed homogeneous approaches are expanded in the compiler and the output is not stored to disk in order to process it by another tool. With respect to our definition of a code generator, where the code is written to a file, these approaches do not belong to it.

1.5 Heterogeneous Code Generators

Heterogeneous code generators are the set of code generators where the meta-language and object language are different. These systems have the advantage that they can realize any possible code generator because they are not limited to any particular object language [111].

A heterogeneous code generator can be implemented based on different implementation approaches. An overview can be found in [120]. We will discuss some commonly used approaches: abstract syntax tree based, printf based generation, term rewriting and text template based.

1.5.1 Abstract Syntax Trees

A technique to generate code for an arbitrary target programming language is to instantiate a tree representation of the output code. This tree is called an *abstract syntax tree*. In Section 2.5 we provide a formal definition of an abstract syntax tree. The code generator instantiates a tree representation of the output code. This tree is transformed to code via a so called *unparser*. The advantages of this approach are that the target programming language is not dependent on the programming language of the generator. The use of a tree ensures syntactical correctness of the output code, when the tree is based on a datatype representing the structure of the target programming language. It is a requirement that the programming language of the code generator is strongly typed in order to ensure that the tree is instantiated in the correct way. The syntactical correctness of the output code depends on the level of detail of the used abstract syntax tree and the correctness of the *unparser*.

An abstract data type, or API, of the target language is required before one can build an abstract syntax tree based generator. This abstract data type can be defined manually, as done in Haskell/DB [79], or an API is off-the-shelf available, like Jenerator for Java [121], or an API can be generated from a grammar, like ApiGen [20].

Haskell/DB provides an abstract data type representing the abstract syntax for SQL queries. The queries are instantiated via this abstract data type and before sending these queries to the database server, the trees are translated into text via a set of print functions.

```
package de.mathema.jenerator.paper;
// imports...
public class HelloJenerator {
  public HelloJenerator() {

    CClass createdClass = new CClass(
         "de.mathema.jenerator.paper", "HelloWorld" );

    CMethod mainMethod = new CMethod( CVisibility.PUBLIC,
                                      CType.VOID, "main" );
    mainMethod.addParameter( new CParameter(
                 CType.user( "String[]" ), "args" ) );
    mainMethod.addToBody( new ClassInstantiation(
              createdClass.getName(), "app", true ) );

    CConstructor cons = new CConstructor(
                                      CVisibility.PUBLIC );
    cons.addToBody( new CCode(
          "System.out.println(\"Hello World!\");" ) );


    createdClass.addConstructor( cons );
    createdClass.addMethod( mainMethod );

    new CodeGenerator().createCode( createdClass );
 }
// main method following here
}
```

Figure 1.4 Jenerator code generator example [121].

```
package de.mathema.jenerator.paper;
public class HelloWorld{
  public HelloWorld(){
    System.out.println("Hello World");
  }
  public void main(String[] args){
    new HelloWorld();
  }
}
```

Figure 1.5 Jenerator output example [121].

A Jenerator example is shown in Figure 1.4. This listing shows how the tree is constructed via instantiating types such as `CClass` and `CMethod`. These types represent nodes in the tree. The hierarchical structure of the tree is visible by the order of calls in the example. First a method node is instantiated and then a class. The result of the code generator is shown in Figure 1.5. A Hello World program is generated.

```
private void genAbstractTypeClass() {
  println("abstract public class " + getClassName()
        + " extends aterm.pure.ATermApplImpl {");
  genClassVariables();
  genConstructor();
  genToTermMethod();
  genToStringMethod();
  genSetTermMethod();
  genGetFactoryMethod();
  genDefaultTypePredicates();

  if (visitable) {
    genAccept();
  }
  println("}");
}

private void genDefaultTypePredicates() {
  Iterator<Type> types = adt.typeIterator();
  while (types.hasNext()) {
    Type type = types.next();
    genDefaultTypePredicate(type);
  }
}

private void genDefaultTypePredicate(Type type) {
  println("  public boolean isSort" +
      TypeGenerator.className(type) + "() {");
  println("    return false;");
  println("  }");
  println();
}
```

Figure 1.6 Printf based generator example (code snippet from ApiGen [20]).

1.5.2 Printf Statements

Printf based generators generate code by printing the code via printf statements to a file or stream. It is possible, just as with abstract syntax trees, to generate code for another target language than the language of the generator. The object code specified in the generator is concrete and therefore better readable. The printf approach does not depend on external libraries or tools, like an unparser, and can be instantly implemented in any programming language that provides print facilities. A drawback of this approach is that it does not provide any guarantees for correctness of the output.

Examples of generators based on the printf approach are ApiGen [20] and NunniFSMGen[4]. Figure 1.6 shows a part of ApiGen.

[4] http://sourceforge.net/projects/nunnifsmgen/ (accessed on November 30, 2010)

1.5.3 Term Rewriting

Term rewriting is a branch of computer science with its foundations in equational logic [8]. It differs from equational logic, since rules are only allowed to replace the left hand side by the right hand side and not vice versa. Let t_1 and t_2 be a term, then a rewrite rule is defined as $t_1 \rightarrow t_2$, where a term matching t_1 is replaced by an instantiation of t_2. t_2 can again contain patterns which match on left hand side of other rules in order to allow further rewriting.

Term rewriting allows to define the projection of input data to source code in a declarative manner by a set of equations, where the left hand side matches on the input data and the right hand side constructs the output source code. The evaluation of rewrite rules enables generation of code in a natural way. This property makes term rewriting a suited solution for generating recursive code, like nested conditionals and loops. Term rewriting offers some abstraction over the implementation of a code generator in an imperative language. Despite this, rewrite rules are necessary to process the structure of the input data. Term rewrite systems come in different flavors, such as processing terms in ELAN [15] and Stratego [118], or such as rewriting concrete syntax in ASF+SDF [13].

Although term rewriting offers advantages, its learning curve is perceived as steep [66]. This experience is amplified in case the terms are based on an abstract syntax tree instead of concrete syntax. Beside that one of the issues of term rewrite systems is the temptation to decompose the object code in almost atomic elements. Understanding the resulting code is hard when all parts of the code are scattered over the rewrite rules. This is also concluded in [108], while investigating XSLT stylesheets [30] for code generation.

We discuss two term rewrite approaches. One applied to terms based on abstract syntax trees, i.e. Stratego, and one using concrete syntax, i.e. ASF+SDF.

A. Stratego

Stratego is a language based on conditional term rewriting. The form of the rewrite rules is l : t_1 `->` t_2 `where` s. The l is the name of the equation, t_1 and t_2 are terms, and s is an optional *conditional*. Normally the rewrite rules are executed via a fixed *strategy*. The Stratego system provides increased flexibility by programmable rewriting strategies, allowing careful control over the application of the rewrite rules.

Before starting to write the transformation one should first define the data types of the input and output. We borrow an example from [116]. The input data consists of a set of *entities*. An entity can contain zero or more properties, which have a name and a type. Figure 1.7 shows a person entity definition.

This person entity can be transformed to a Java class via a rewrite rule. Since Stratego is based on terms, the right hand side of the rule is an abstract datatype

```
Entity("Person",
  [ Property("fullname", SimpleSort("String")),
    ...
    Property("homepage", SimpleSort("String"))
  ]
)
```

Figure 1.7 Data structure containing a *Person* entity definition.

```
entity-to-class :
  Entity(x, prop*) ->
  ClassDec(
    ClassDecHead(
      [MarkerAnno(TypeName(Id("Entity"))), Public()]
      , Id(x)
      , None(), None(), None()),
    ClassBody(
      [ConstrDec(
         ConstrDecHead([Public()],None(),Id(x),[],None()),
         ConstrBody(None(), []))
      ])
  )
```

Figure 1.8 Rewrite rule to create a class based on an entity definition.

representing the hierarchical structure of the output code. This code must be *unparsed* to get a concrete syntax representation. *Unparsing* is the conversion of abstract syntax tree to concrete syntax. An example of a Stratego rewrite rule which matches the data structure of Figure 1.7 is given in Figure 1.8.

The rewrite rule matches on the `Entity` term on the left hand side and for each entity it instantiates a Java class. The *variable* `x` is used to parameterize the class with the name of the given entity. In case of the example person entity, `x` will become `Person`. The result of the equation is a tree, which is unparsed and shown in Figure 1.9. The properties of the `Entity` are ignored by this example rewrite rule and as a result not shown in the output listing.

This example shows the application of Stratego in a code generation context.

```
@Entity
public class Person{
 public Person() {    }
}
```

Figure 1.9 Result code of the rewrite rule.

```
entity-to-class :
  Entity(x_Class, prop*) ->
  |[
     @Entity
     public class x_Class{
       public x_Class() { }
     }
  ]|
```

Figure 1.10 Rewrite rule using concrete syntax.

The use of abstract syntax terms is not easy to read and write. Stratego offers a mechanism to use *concrete* syntax instead of abstract syntax in the rewrite rules [119]. The previous defined rule can be implemented using concrete syntax as shown in Figure 1.10. Note that the identifier `x_Class` is automatically recognized by the Stratego parser as a metavariable.

B. ASF+SDF

Another example of a rewrite system is ASF+SDF [13]. Where Stratego is based on manipulating abstract syntax trees, ASF+SDF is based on concrete syntax rewriting. ASF+SDF also ensures that the output code must conform to a predefined grammar and enforces syntax correctness of the generated code.

ASF+SDF is based on two formalisms. The first is SDF, an acronym for Syntax Definition Formalism, which is a formalism to specify the syntaxes of (programming) languages and to define the function signatures for ASF. We will discuss SDF in Chapter 2 in detail. The Algebraic Specification Formalism, ASF, is a rewriting formalism. An ASF specification is a collection of equations. The equations have a left hand side (lhs) which matches on patterns defined in SDF and a right hand side (rhs) specifying the result pattern also defined in SDF. The form of these equations is $s = t$, where s and t are concrete syntax terms. Furthermore ASF supports conditional equations with a set of conditions, which should succeed before the equation is reduced. The form of conditional equations is $s_1 = t_1$ `,...,` $s_n = t_n$ `===>` $s = t$, where all variants of s and t are concrete syntax terms. During interpretation of the conditional equation, first the equations before the arrow sign are evaluated and if all succeed, the equation after the arrow sign is evaluated.

Figure 1.11 shows the Stratego example expressed in ASF. The example defines the equation for the function `generate: Term -> Java` using a conditional rewrite rule. The equation after the arrow defines the transformation of the input to a piece of Java. Its left hand side is the `generate` function and includes a match pattern for the input term. The match pattern includes variables, recognizable by the `$` prefix. Variables must be manually declared in the SDF grammar, per syntactical sort. The right hand side defines the resulting Java code. The Java code contains a variable `$id`, which is not bound in the lhs pattern. ASF ensures syntax correctness and the variables bound in the lhs are not of the correct syntactical type. The value for `$id` is obtained by casting the `$class` variable via the function `str_to_id`. This casting is specified before the arrow, in case the casting is not successful, the equation will not be applied.

C. RASCAL

This section will briefly discuss RASCAL [72], a domain specific language for source code analysis and manipulation. It is designed to cover the range

```
[entity-to-class]
  $id := str_to_id( $class )
  ====>
  generate(Entity($class , $props)) =
   @Entity
   public class $id{
      public $id() { }
   }
```

Figure 1.11 Stratego example expressed in ASF.

of applications from pure analyzes to pure transformations and everything in between. There are a lot of libraries, tools and languages available for software analysis and transformation, but integrated facilities that combine both domains are scarce. Tools either specialize in analysis or in transformation, but not in both. RASCAL combines both domains by providing high-level integration of code analysis and manipulation on the conceptual, syntactic, semantic and technical level.

RASCAL is grafted on different already existing languages and paradigms. Relational calculus is provided for analysis, syntactical analysis is based on SDF and term rewriting rules are inspired by ASF. The values are represented as ATerms. Furthermore, concepts of Haskell, Java and Ruby are used.

In short, the requirements for RASCAL resulted in the following features. It supports the common basic datatypes, such as boolean and strings. Also lists and other collection data types are supported, including type parametric polymorphism. Since source-to-source transformation requires concrete syntax patterns, the types of (parse) trees generated by a given grammar are also first class RASCAL types. Different pattern matching algorithms, like string matching based on regular expressions, are provided. Fully typed rewrite rules as defined by ASF+SDF are available. RASCAL also provides comprehensions to express the calculation of relation analyzes in a concise way. Finally switch statements and exception handling as can be found in Java, are integrated in RASCAL.

RASCAL is suited for code analysis and code synthesis, i.e. code generation. If source code is to be generated, RASCAL provides various options. As RASCAL is a language based on different other languages and paradigms, it supports different approaches to generate code. In essence, all the approaches discussed in this section are available in RASCAL. First, it is possible to use print statements with embedded variables. Second, rewrite rules can be used to instantiate abstract syntax trees, which can be pretty printed to strings. Third, just as ASF, concrete syntax rewrite rules can be used. Finally, text templates, as will be discussed in Section 1.5.4 are available. The safety levels of these different approaches are equal to the discussed variants.

As templates is the topic of this thesis, Figure 1.12 shows an example of a

```
module demo::StringTemplate

import String;

public str capitalize(str s) {
  return toUpperCase(substring(s, 0, 1))
                         + substring(s, 1);
}

public str genClass(str name,
                    map[str,str] fields) {
  return "
    public class <name> {
     <for (x <- fields) {
       str t = fields[x];
       str n = capitalize(x);>
       private <t> <x>;
       public void set<n>(<t> <x>) {
         this.<x> = <x>;
       }
       public <t> get<n>() {
         return <x>;
       }
     <}>
    }
  ";
}
```

Figure 1.12 Example of a RASCAL template (from [71]).

string template in RASCAL. A template in RASCAL is a string with meta syntax between less-than and greater-than symbols. The meta syntax is replaced during evaluation of the template. Since the template is a string, no guarantees of the syntax correctness are given. It can be called using the call `genClass("Person", fields)` and the map:

```
public map[str, str] fields = (
  "name" : "String",
  "age" : "Integer",
  "address" : "String"
);
```

The output of the template using this input data is a class `Person` having three fields including getter and setter methods. In Section 1.5.4 text templates are discussed in detail.

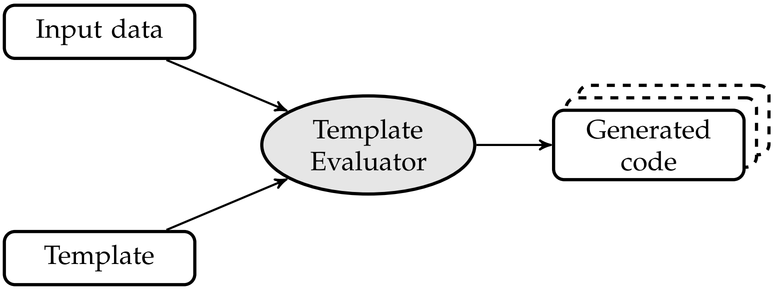


Figure 1.13 Template based generator.

1.5.4 Text Templates

A text template system is the last discussed approach to implement code generators. This approach is known from its use for instantiating HTML in dynamic websites or web applications [34]. As a result numerous template evaluators are designed for instantiating HTML in web applications. Besides generating HTML, text templates can be used for generating all kinds of unstructured text, like e-mails or code. Figure 1.13 shows the four components involved in a text template based generator. These components are input data, template, template evaluator and generated code.

Since the literature does not provide a formal definition of a template we start with the informal operational definition provided by Parr [92]:

> "A template is an output document with embedded actions which are evaluated when rendering the template."

Following this definition a template is a text document that *can* contain *placeholders*. A placeholder is a (syntactical) entity, indicating a missing piece of text. It contains some action, or expression, declaring how to obtain a piece of text to replace it. In the most basic form for human interpretation this action is a sequence of dots, or for automatic processing the action can range from labels to a piece of code belonging to a general purpose programming language.

More formally, a template is a $(text|placeholder)+$ pattern, i.e. an arbitrary non-empty sequence of text fragments and placeholders. The *text* is the fixed part of the template and is one-to-one copied to the output document. The *placeholder* represents a non-complete part of the text.

In case of templates, the placeholders are the meta code expressed in a metalanguage. Informally, a metalanguage automatically originates at the moment placeholders are introduced in a piece of text or code. This is not only applicable for computer languages, but also happens in natural languages. For example, when one writes a generic agreement or letter where the names are replaced by a sequence of dots (...). For each new client the dots must be filled with specific information. The sequence of dots is for human interpretation, where someone can replace the dots using the context of the document and environment. Considering a template for source code processed automatically, the metalanguage must have formal semantics and should contain explicit instructions.

Automatic processing of templates is performed by a so called *template evaluator*. This is an application which interprets a template in order to generate text. It searches for placeholders, executes the specified action and replaces the placeholders to complete the output text.

The combination of template and template evaluator constitutes a code generator. The template contains the application specific part of the code generator, while the template evaluator is the generic part of the code generator. The generic part of the code generator is amongst others responsible for handling the output text, input data processing and other administration tasks such as directory creation and file creation. Separating the output code patterns from the generator code improves the understandability of what output the code generator produces during generation.

The separation between generator code and output code fragments reduces the complexity compared to the abstract syntax tree approach or printf approach, where generator code and output code fragments are mixed. In these approaches the generator logic is mixed with the output code, while the text template approach separates these two aspects of a generator. Furthermore, the code patterns in a text template are written in concrete syntax.

Text templates can be used to generate code for every target language. It is a common pattern for building webpage generators [34], but it is also used in many code generator frameworks such as model driven engineering tools like openArchitectureWare[5]. A few examples of text template evaluators are Apache Velocity[6], StringTemplate [92], ERb[58], Java Server Pages[7], FreeMarker[8] and Smarty[9].

Figure 1.14 shows a template for the Apache Velocity template evaluator. The `$` signs are used for Java object references to obtain values. Instructions for the template evaluator are prefixed by a `#`.

This example shows the loop construction `#foreach`, which loops over a list of objects and evaluates the body of the loop where the context is set to the current processed element of the list. The `#set` directive is used for setting a

[5] http://www.openarchitectureware.org (accessed on November 30, 2010)

[6] http://velocity.apache.org (accessed on November 30, 2010)

[7] http://java.sun.com/products/jsp/ (accessed on November 30, 2010)

[8] http://freemarker.org (accessed on November 30, 2010)

[9] http://www.smarty.net (accessed on November 30, 2010)

```
/*
* Created on $newDate
* generated by a FUUT-je application using
* Velocity templates
*/
package $package;
public class $class.Name {
  #foreach($att in $class.Attributes)
  protected $att.Type ${att.Name};
  #end

  #foreach($att in $class.Attributes)
  /**
  * @return Returns the $att.Name
  */
  #set( $uName = "${ft.capFirst($att.Name)}")
  public String get${uName}() {
    return $att.Name;
  }

  /**
  * @param $att.Name The $att.Name to set.
  */
  public void set${uName}(String ${att.Name}) {
    this.${att.Name} = $att.Name;
  }
  #end
}
```

Figure 1.14 Example of an Apache Velocity template [89].

value. In this example it is used to upper case the first character of the identifier of the field name to let the output code comply with the Java coding standards.

Technique	Advantages	Disadvantages
Homogeneous	-No external tools -Type safety	-Mono lingual
Abstract Syntax Tree	-Heterogeneous -Syntax safety	-Output code not concrete -Abstract Data Type required of object language -Unparser required
Printf	-Heterogeneous -Output code concrete	-No safety
Term rewriting	-Heterogeneous -Syntax safety	-Complex technique
Text Templates	-Heterogeneous -Output is concrete -Generic generator code separated	-No safety

Table 1.1 Overview advantages and disadvantages code generator implementation approaches.

1.6 Conclusions

The context of this thesis is raising the level of abstraction during programming. One of the issues in raising the level of abstraction is that models at a higher abstraction level must be translated to lower level implementations. This translation can, among other solutions, be performed by a code generator.

The reasons to choose for code generation can be categorized in three topics: Using an object language independent of the metalanguage, raising the level of abstraction, or improving the execution performance. Different implementation approaches of a code generator are discussed. The presented code generator implementation approaches all have their advantages and limitations. Table 1.1 provides an overview of them.

A homogeneous system is superb in terms of syntax safety and type safety, because it is a language feature. This is immediately also the drawback for homogeneous systems. It is only possible to generate code for the language itself. Further the homogeneous approach can only be used when the output language can express computations, and thus can act as both metalanguage and object language. This is not always the case, for instance HTML cannot be used as metalanguage, since it cannot express behavior.

When it is necessary to generate code for another language than the metalanguage, one can use abstract syntax trees, printf statements, term rewriting systems, or text templates. The approach using abstract syntax trees can result in a code generator, whose output code is syntactical correct. The trade-off is the complexity of building and maintaining code generators based on this approach. The object code inside the generator is not concrete. It is hard to read the object code, since it is encapsulated in a data structure representing the abstract syntax of the metalanguage. A detailed knowledge of the object language grammar structure is necessary. Finally having an abstract data type is not sufficient, an unparser must also be available to transform the instantiated syntax tree into text.

Using printf statements removes the problem of *invisible* object code, since the object code is concrete. This approach can be used in most languages without the necessity of external libraries, because printf statements, or equivalent, are most times directly supported by the language. The drawback of a printf based generator is its simplicity. The output code written in the printf statements is handled as strings. Syntax errors are not detected by the generator or by the compiler of the metalanguage.

Term rewriting allows to define a code generator in a declarative manner by a set of equations, where the left hand side matches on the input data and the right hand side constructs the output source code. Some term rewriting systems, such as ASF+SDF, offer an approach which allows concrete syntax for the object code. ASF+SDF uses grammars to parse all code in the rewrite rules; terms may only be replaced by other terms of the same syntactical type, as a result syntactical correctness is guaranteed. Unfortunately, term rewriting based code generators do not remove the entangling between the object code and the code processing the input data and file manipulation.

The text template approach offers abstraction of most of the generic generator code, which is captured in the template evaluator. Only small chunks of meta code in the template are necessary to instruct the template evaluator. The result is that the template looks very similar to the output code it instantiates. The drawback of text templates is already in the name. The evaluator of text templates does not consider the object code and handles it as a sequence of characters and thus no guarantees can be given that the output code is syntactically correct.

1.7 Problem Statement

The subject of this thesis is template based code generators. The summary in the previous section brings us to the central research question of this thesis:

> **Central research question:** *How can the quality of template based code generators be improved?*

Quality is a broad notion and we need to specify it in more concrete requirements. We are interested in the technical quality of template based code generators, i.e. the correctness of the output and the computational power of the template. We already concluded that text templates offer no guarantees that the output code is correct. Our aim is to increase the guarantees for correct output and find errors as early as possible, not at the moment the generated code is compiled or interpreted. Text templates are widely used, for example in web applications, as a result of their perceived usability. Our aim is not to affect this level of usability.

Improving the technical quality makes debugging less hard, since errors are detected earlier and the origin of the error can be better determined. It also reduces the chance that generated code or a code generator shipped to a client contains bugs. Beside the use in the development process of software, the presented solution offers also increased safety of applications generating code on the fly.

This thesis consists of an introduction followed by seven chapters and a conclusion. Each of these chapters answers a research question in the context of the central theme and they are arranged in order of dependency. Chapter 2 presents a literature study on the topic of formal languages. It provides the basic definitions and notations used in this thesis.

Chapter 3 discusses the relation between different grammar classes in order to obtain the requirements that a metalanguage for code generators should fulfill. The research question for this chapter is:

> **Research question 1:** *What are the requirements of a metalanguage for code generators?*

We use the theory on formal languages to specify the requirements and properties a metalanguage should satisfy. The relations between concrete syntax, abstract syntax, parser, unparser and their underlying grammars are discussed.

The research question for Chapter 4 is:

> **Research question 2:** *What is a minimal metalanguage to facilitate templates?*

Using the requirements specified in Chapter 3 a metalanguage for templates is defined. Our aim is not to develop a completely new programming language and we use existing theory on programming languages to define our metalanguage. We show that our metalanguage meets the requirements presented in Chapter 3 and is still limited to enforce separation of model and view. This chapter finishes with a comparison of different related template evaluators used in the industry and the scientific world.

Given the metalanguage, Chapter 5 discusses the research question:

> **Research question 3:** *How can we check the syntax of the object language and metalanguage in a template simultaneously?*

The relation between the grammars of the object language and the metalanguage is presented. It is used to specify a template grammar containing rules for both object language as well as metalanguage. Having a grammar containing rules for both languages enables checking of both languages simultaneously by a parser. The presented approach is object language parametric and every object language can be extended with a metalanguage as long the object language comes with a context-free grammar.

Checking only the syntax of the template is not sufficient to guarantee that the output of the template evaluator produces code without syntax errors. Therefore the research question for Chapter 6 is defined:

> **Research question 4:** *How can we guarantee that a template always generates code without syntax errors?*

Parsing a template using the grammar as defined in Chapter 5 results in a parse tree containing nodes for both languages. We present an approach to evaluate the meta code and replace it by object language sub parse trees ensuring that the final output of the evaluator is a valid object language parse tree. The approach is based on a single tree traversal and uses the object language grammar to verify that the meta code is substituted by a valid piece of object code. One of the issues of templates and parsing them is to deal with ambiguities. We present a generic approach to handle ambiguities introduced by combining the object language grammar and metalanguage grammar. The ideas are implemented in a template evaluator called *Repleo*.

The implementation Repleo enables to answer the research question of Chapter 7:

> **Research question 5:** *Can we use the approach to implement real world code generators?*

This chapter shows a number of case studies using Repleo. Repleo is used for code generation in different application domains; the generation of web applications, data structures and state machines. Furthermore we show how the syntax checking of templates results in protection against injection attacks in web applications.

This thesis mainly concerns the syntactical correctness of templates and generated code. Beside the syntax, also the static semantic correctness of a template and the generated code are important. In Chapter 8 we present our results on the topic of the research question:

> **Research question 6:** *Can we determine static semantic errors in meta code and object code without generating the actual code?*

We define checks for the metalanguage and the object language of templates. A prototype implementation for PicoJava templates is used to validate our ideas of checking templates. The aim is to reuse as much as possible of the implementation of the static semantic checker of the object language. We conclude with suggestions for future work.

This thesis ends with a discussion of its contributions and with final conclusions.

2

Preliminaries

This chapter provides basic notations, definitions and properties needed throughout this thesis. We take our notations and definitions from the literature [31, 33, 39, 59]. This chapter can be skipped on first reading and referred to when necessary.

2.1 Basic Definitions and Notations

We use formal language theory to study templates and describe their syntax. This section provides definition for common concepts of formal languages as we use it throughout the thesis. Our definitions are based on automata and language theory [31, 33, 39, 59]:

- A *symbol* is a syntactic entity without any meaning.
- An *alphabet* is a finite non-empty set of symbols.
- The *rank* of a symbol is the number of branches leaving it.
- A *ranked alphabet* is a pair of an alphabet and ranking functions, where the rank function maps a symbol in the alphabet to a single rank.
- A *string* is a finite sequence of symbols chosen from the given alphabet.
- A *language* is the set of all strings belonging to an alphabet, including the empty string. This definition for "language" may be strange with the intuitive notion of what language means. However, a language can indeed be seen as a set of strings, for example C, where the *well-formed* programs are a subset of the possible strings that can be formed by the alphabet of that language. The alphabet of the C language is formed by symbols existing of keywords, brackets, operators, natural numbers and the English alphabet. The set of sentences produced by these symbols

includes the set of well-formed C programs. The set of well-formed C programs is defined by a context-free grammar for the alphabet and semantic rules.

- ◇ A *terminal* symbol is a symbol from which sentences are formed and it occurs literally in a sentence [5], i.e. terminal-symbols are elements of the alphabet.
- ◇ A *nonterminal* symbol is a variable representing a sequence of symbols and it can replace a string of terminal symbols or a string existing of a combination of terminal and nonterminal symbols [5].

We use the concept of *tree* in this thesis. The trees meant in this thesis are defined by a number of properties. Based on graph terminology, a tree is finite (finite number of nodes and branches), directed (top-down), rooted (there is one node, the root, with no branches entering it), ordered (the branches leaving a node are ordered left to right) and labeled (the nodes are labeled with symbols from a given alphabet) [39]. The following terminology will be used:

- ◇ A *leaf* is a node with rank 0.
- ◇ The *top* of a tree is its root.
- ◇ A *path* through a tree is a sequence of nodes connected by branches (“leading downwards”).
- ◇ A *subtree* of a tree is a tree determined by a node together with all (the subtrees of) its children.

We present in this thesis mathematical definitions and theorems based on formal language theory. We use a naming convention to denote sets, relations, functions etc. based on the names as they occur in the literature. We have chosen the names, so they are not ambiguous. The following list presents the names used and basic definitions.

- ◇ k, i, j, p and r are used for integer variables.
- ◇ Σ is used to denote an alphabet. For example $\Sigma = \{0, 1\}$, the binary alphabet and $\Sigma = \{a, b, \ldots, z\}$ the set of all lower-case letters.
- ◇ Σ^* denotes all strings over an alphabet Σ.
- ◇ σ and c for alphabet symbols.
- ◇ N for nonterminal alphabets.
- ◇ n, A for nonterminal symbols of N.

- ◇ y for sequences of alphabet symbols combined with nonterminal symbols (i.e. strings, elements of $(N \bigcup \Sigma)^*$).
- ◇ z for alphabet symbols or nonterminal symbols ($z \in (N \bigcup \Sigma)$).
- ◇ ϵ is used for the empty string or null value.
- ◇ $\mathcal{L}$ for languages.
- ◇ s for sentences of a language $\mathcal{L}$ defined by Σ^*.
- ◇ r denotes the rank of a symbol and $r \in \mathbb{N}_0$ and is defined by the ranking function $r_\sigma = rank(\sigma)$ where $\sigma \in \Sigma$. Each symbol in a ranked alphabet (see Definition 2.1.1) has a unique rank.
- ◇ Σ_r for the set of symbols of rank r.
- ◇ $Tr(\Sigma)$ denotes the set of trees over a ranked alphabet Σ, i.e. Σ including a set of ranking functions over Σ.
- ◇ t for trees, see Definition 2.1.3.
- ◇ a for alphabet symbols with rank 0 ($a \in \Sigma_0$).
- ◇ f for alphabet symbols with rank greater than 0 ($f \in \Sigma_r,$ where $r > 0$).
- ◇ X is a set of symbols called variables and we assume that the sets X and Σ_0 are disjoint.
- ◇ x is a variable $x \in X$ and is not used for integer values.
- ◇ G for grammars.

Definition 2.1.1. (Ranked alphabet) [39]. An alphabet Σ is said to be ranked if for each nonnegative integer k a subset Σ_k of Σ is specified, such that Σ_k is nonempty for a finite number of k's only, and such that $\Sigma = \bigcup_{k \geqslant 0} \Sigma_k$. If $\sigma \in \Sigma_k$, then we say that σ has rank k.

Example 2.1.2. (Ranked alphabet) [39]. The alphabet $\Sigma = \{a, b, +, -, *\}$ is made into a ranked alphabet by specifying $\Sigma_0 = \{a, b\}$, $\Sigma_1 = \{-\}$ and $\Sigma_2 = \{+, *\}$.

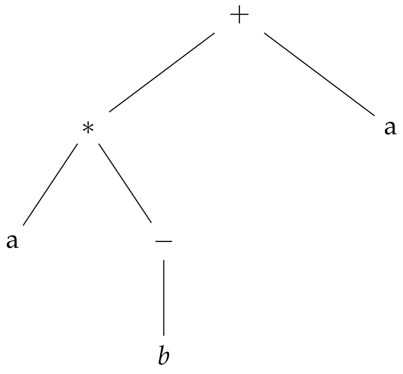
Definition 2.1.3. (Tree) [39]. Given a ranked alphabet Σ, the set of trees over Σ, denoted by $Tr(\Sigma)$ is the language over the alphabet $\Sigma \cup \{[,]\}$, where $\Sigma \cap \{[,]\} = \emptyset$, defined inductively as follows.

(i) If $\sigma \in \Sigma_0$, then $\sigma \in Tr(\Sigma)$.

(ii) For $k \geqslant 1$, if $\sigma \in \Sigma_k$ and $t_1, \ldots, t_k \in Tr(\Sigma)$, then $\sigma[t_1 \ldots t_k] \in Tr(\Sigma)$.

Example 2.1.4. (Tree) [39]. Consider the ranked alphabet of Example 2.1.2. Then $+[*[a-[b]]a]$ is a tree over this alphabet, intuitively *representing* the tree:

$+[*[a-[b]]a] =$



Which on its turn represents the concrete expression $(a*(-b))+a$.

Definition 2.1.5. (Linear tree) [39]. A term $t \in Tr(\Sigma \cup X)$ is linear when each variable is at most used once in t.

Definition 2.1.6. (Substitution) [33]. A substitution (respectively a ground substitution) m is a mapping from X into $Tr(\Sigma \cup X)$ (respectively into $Tr(\Sigma)$) where there are only finitely many variables not mapped to themselves. The domain of a substitution m is the subset of variables $x \in X$ such that $m(x) \neq x$. The substitution $\{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k\}$ maps $x_i \in X$ on $t_i \in Tr(\Sigma \cup X)$, for every index $1 \leq i \leq k$.

Substitutions can be extended to $Tr(\Sigma \cup X)$ in such a way that:

$$\forall f \in \Sigma_r, \forall t_1, \ldots, t_r \in Tr(\Sigma \cup X) \;\; m(f(t_1, \ldots, t_r)) = f(m(t_1), \ldots, m(t_r)).$$

Example 2.1.7. (Substitution) [33]. Let $\Sigma = \{f(,,), g(,), a, b\}$ and $X = \{x_1, x_2\}$. Let us consider the term $t = f(x_1, x_1, x_2)$. Let us consider the ground substitution $m = \{x_1 \leftarrow a, x_2 \leftarrow g(b,b)\}$ and the substitution $m' = \{x_1 \leftarrow x_2, x_2 \leftarrow b\}$. Then $m(t) = t\{x_1 \leftarrow a, x_2 \leftarrow g(b,b)\} = f(a,a,g(b,b))$ and $m'(t) = t\{x_1 \leftarrow x_2, x_2 \leftarrow b\} = f(x_2, x_2, b)$.

Definition 2.1.8. (Tree homomorphism) [33]. Let Σ and Σ' be two not necessarily disjoint ranked alphabets. For each $k > 0$ such that Σ contains a symbol of rank k, we define a set of variables $X_k = \{x_1, \ldots, x_k\}$ disjoint from Σ and Σ'.

Let h_Σ be a mapping which, with $\sigma \in \Sigma$ of rank k, associates a term $t_\sigma \in Tr(\Sigma', X_k)$. The tree homomorphism $h : Tr(\Sigma) \to Tr(\Sigma')$ is determined by h_Σ as follows:

- ◇ $h(a) = t_a \in Tr(\Sigma')$ for each $a \in \Sigma$ of rank 0,
- ◇ $h(\sigma(t_1, \ldots, t_n)) = t_\sigma\{x_1 \leftarrow h(t_1), \ldots, x_k \leftarrow h(t_k)\}$
  where $t_\sigma\{x_1 \leftarrow h(t_1), \ldots, x_k \leftarrow h(t_k)\}$ is the result of applying the substitution $\{x_1 \leftarrow h(t_1), \ldots, x_k \leftarrow h(t_k)\}$ to the term t_σ.

h_Σ is called a *linear tree homomorphism* when no t_σ contains two occurrences of the same x_k. Thus a linear tree homomorphism cannot copy trees.

Example 2.1.9. (Tree homomorphism) [33]. Let $\Sigma = \{g(,,), a, b\}$ and $\Sigma' = \{f(,), a, b\}$. Let us consider the tree homomorphism h determined by h_Σ defined by: $h_\Sigma(g) = f(x_1, f(x_2, x_3)), h_\Sigma(a) = a, h_\Sigma(b) = b$. For instance, we have: If $t = g(a, g(b, b, b), a)$, then $h(t) = f(a, f(f(b, f(b, b)), a))$.

2.2 Context-free Grammars

This thesis will focus on the generation of sentences of languages aimed to express programs executed or interpreted by a computer. The rules for constructing valid sentences of these languages can be specified by context-free grammars. The *syntax*[1] of a language is its valid set of sentences. Compilers or interpreters for most programming languages are based on LL or LR parsers. LL or LR parsers can handle a subset of the context-free grammars, which implies that these programming languages are context-free languages. A context-free language $\mathcal{L}(G)$ is specified by a context-free grammar G. A sentence belonging to the set of sentences specified by a context-free grammar is called a *well-formed* sentence. The context-free grammar is defined as follows [51]:

Definition 2.2.1. (Context-free grammar). A context-free grammar (CFG) is a four-tuple $\langle \Sigma, N, S, Prods \rangle$ where

Σ is a finite set of terminal symbols, i.e. the alphabet.

N is a finite set of nonterminal symbols and $N \bigcap \Sigma = \emptyset$.

S is the start symbol, or axiom, and $S \in N$.

$Prods$ is a finite set of production rules of the form $n \rightarrow y$ where $n \in N$ and $y \in (N \bigcup \Sigma)*$.

Each context-free grammar G_{cfg} can be transformed into a Chomsky normal form without changing the language generated by that grammar [60]. A context-free grammar of the Chomsky normal form only contains rules of the forms:

1. $A \rightarrow \epsilon\{c\}$, where $A \in N$ and A is the start symbol;
2. $A \rightarrow s\{c\}$, where $A \in N$ and $s \in \Sigma^*$;
3. $A \rightarrow n_1 n_2\{c\}$, where $A, n_1, n_2 \in N$.

[1] The syntax rules do not specify the meaning of a sentence, as a result a syntactical correct sentence can be nonsense.

Example 2.2.2 shows a context-free grammar definition for a language based on boolean algebra.

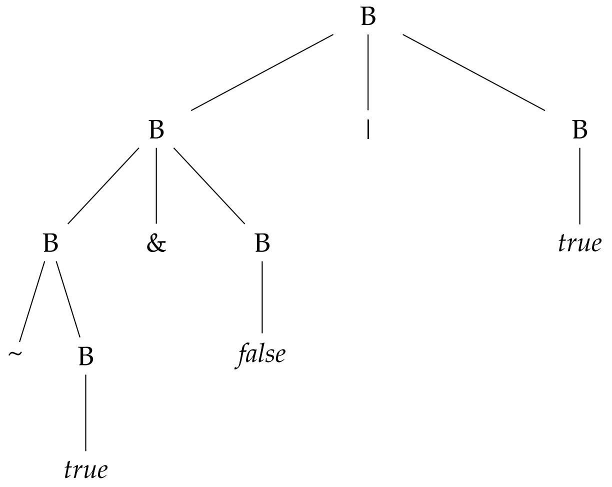
Example 2.2.2. (Context-free grammar). Let G_{bool} be a context-free grammar with, $\Sigma = \{\ \sim, \&, |, (,), \mathit{true}, \mathit{false}\ \}$, nonterminals $N = \{\mathrm{B}\}$, start symbol $S = \mathrm{B}$ and rules

$$
\begin{array}{l}
\mathit{Prods} = \{ \\
\qquad \mathrm{B} \rightarrow \text{“}\sim\text{”}\ \mathrm{B}, \\
\qquad \mathrm{B} \rightarrow \mathrm{B}\ \text{“}\&\text{”}\ \mathrm{B}, \\
\qquad \mathrm{B} \rightarrow \mathrm{B}\ \text{“}|\text{”}\ \mathrm{B}, \\
\qquad \mathrm{B} \rightarrow \text{“(”}\ \mathrm{B}\ \text{“)”}, \\
\qquad \mathrm{B} \rightarrow \text{“}\mathit{true}\text{”}, \\
\qquad \mathrm{B} \rightarrow \text{“}\mathit{false}\text{”} \\
\ \}
\end{array}
$$

A context-free grammar defines a set of sentences, i.e. the language $\mathcal{L}(G)$, where for each $s \in \Sigma$ a derivation exists $S \xRightarrow[G]{*} s$. If a sentence belongs to $\mathcal{L}(G)$ a parse tree can be constructed using the grammar. This tree is derived by applying the production rules of the grammar to construct the sentence and it is called the *parse tree* [5]. Example 2.2.3 shows a parse tree derived from a sentence of $\mathcal{L}(G_{bool})$.

Example 2.2.3. (Parse tree). Let s be `~true & false | true`, the parse tree of s using the grammar G is:

$d_G(s) =$



Note: The language of Example 2.2.2 is ambiguous and we assume that the operators are parsed left-associative.

A parse tree represents the hierarchical structure of the sentence expressed by the production rules of its grammar. Normally such parse trees are automatically constructed from a given sentence when a parser is used based on the grammar. A parser can, for instance, use algorithms like LL [3] and LR [94].

The parse tree contains enough information to restore the original sentence. Consider the parse tree of Example 2.2.3, and read the leaves from left to right, the original sentence is visible. The *yield* function reconstructs the original string of a parse tree. It traverses a parse tree in order to compute the original sentence from it by concatenating the leaves (taking the leaf symbols as letters) from left to right.

Definition 2.2.4. (Yield). The *yield* function is defined by the following two rules:

- ◇ $\mathit{yield}(a) = a$ if $a \in \Sigma_0$;
- ◇ $\mathit{yield}(f(t_1, \ldots, t_k)) = \mathit{yield}(t_1) \cdot \ldots \cdot \mathit{yield}(t_k)$ if $f \in \Sigma_k$ and $t_i \in \mathit{Tr}(\Sigma \cup N)$, where $\cdot$ denotes the string concatenation.

2.3 Regular Tree Grammars

A *regular tree language* is a set of trees generated by a *regular tree grammar*. The definition of regular tree grammars is [33]:

Definition 2.3.1. (Regular tree grammar). A regular tree grammar (RTG) is a four-tuple $\langle \Sigma, N, S, \mathit{Prods} \rangle$, where:

Σ is a finite set of terminal symbols with rank $r \geq 0$.

N is a finite set of nonterminal symbols with rank $r = 0$ and $N \cap \Sigma = \emptyset$.

S is a start symbol and $S \in N$.

Prods is a finite set of production rules of the form $n \to t$, where $n \in N$ and $t \in \mathit{Tr}(\Sigma \cup N)$.

Example 2.3.2 shows a regular tree grammar (taken from [31]).

Example 2.3.2. (Regular tree grammar). Let G be the regular tree grammar with $\Sigma = \{a(,), b(), c\}$, nonterminals $N = \{E, W\}$, start symbol E, and rules

$$
\begin{aligned}
\mathit{Prods} = \{ & \\
& \mathrm{E} \to \mathrm{W}, \\
& \mathrm{W} \to \mathrm{b(W)}, \\
& \mathrm{W} \to \mathrm{b(a(c,c))} \\
\} &
\end{aligned}
$$

The language of this grammar is

$$\mathcal{L}(G_{rtg}) = \{b(a(c,c)), b(b(a(c,c))), b(b(b(a(c,c)))), ...\}.$$

The parse steps of the term $b(b(a(c,c)))$ are $E \Rightarrow W \Rightarrow b(W) \Rightarrow b(b(a(c,c)))$, where $\Rightarrow$ is a derivation step.

Regular tree languages have a number of properties [31], the one being important for us is *recognizability* of regular tree languages. Recognizable tree languages are the languages recognized by a finite tree automaton. Regular tree languages are recognizable by (non)-deterministic bottom up finite tree automata and non-deterministic top-down tree automata [33]. The set of languages recognizable by deterministic top-down tree automata is limited to the class of path-closed tree languages, a subset of regular tree languages.

2.4 Relations between CFL and RTL

A number of relations can be defined between context-free languages and regular tree languages [31]. First a regular tree language is a generalization of context-free languages as any string can indeed be seen as a non-branching tree.

Furthermore, a tree can be represented as a term. These terms can be parsed using a context-free grammar, since printing a (sub)tree to text does not depend on the sibling nodes of that (sub)tree. This context-free grammar of our term representation is given in Figure 2.4. For example the tree of Example 2.2.2 can be represented by the term

$$B(B(B(\sim, B(\text{true})), \&, B(\text{false})), \mid, B(\text{true})).$$

More interesting for code generation is the relation between regular tree languages and the parse trees of context-free languages. The *parse* function takes a string and a grammar and returns the parse tree of that string when the string can be produced by that grammar. The way parser algorithms create a parse tree shows regularity, which suggests that the parse trees are indeed regular. A proof that the set of parse trees of a context-free grammar is a regular tree language can be found in [33]. The following definition shows the derivation of the regular tree grammar $\mathcal{L}(G_{pt})$ defining the set of parse trees of a context-free grammar $\mathcal{L}(G_{cfg})$.

Definition 2.4.1. (Regular tree grammar for parse trees) [33]. Let $G_{cfg} = \langle \Sigma, N, S, Prods \rangle$ be a context-free grammar. The regular tree grammar $G_{pt} = \langle \Sigma', N', S', Prods' \rangle$ defining the parse trees of G_{cfg} is derived by the following rules:

- The start symbol of both grammars is equal: $S = S'$,

- The set of nonterminals of both grammars is equal: $N = N'$,

- The alphabet of G_{pt} is derived by the following rule:
  $\Sigma' = \Sigma \bigcup \{\epsilon\} \bigcup \{(A,k) | A \in N, \exists A \to y \in \mathit{Prods}$ with y of length $k\}$. Normally in parse trees a symbol can have a different number of children, when alternative production rules have a different pattern length. In tree languages a symbol must have a fixed rank, so we introduce a symbol (A,k) for each $A \in N$ such that there is a rule $A \to y$ with y of length k.

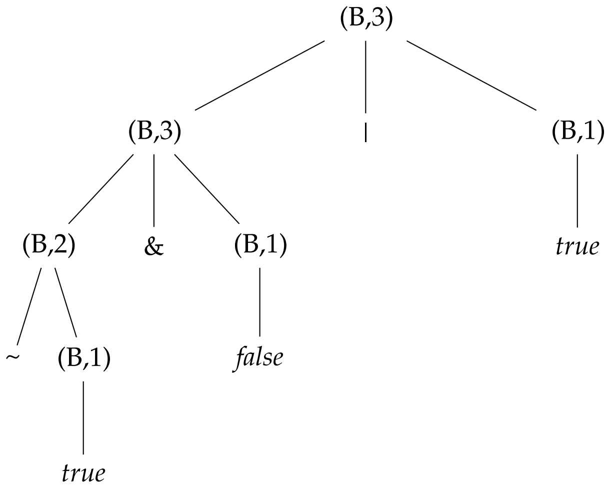
- The set of productions Prods' of G_{pt} is derived by the following rules:

  if $A \to \epsilon \in \mathit{Prods}$ then $A \to (A,0)(\epsilon) \in \mathit{Prods}'$.

  if $(A \to a_1 \dots a_k) \in \mathit{Prods}$ then $A \to (A,k)(a_1, \dots, a_k) \in \mathit{Prods}'$.

Example 2.4.2. (Regular tree grammar for parse trees). Since normally in parse trees a symbol can have different number of children, we give an updated version of the parse tree displayed in Example 2.2.3:

$d_G(s) =$



Using the definition given above we can derive the G_{pt} defining the language of parse trees of G_{bool}. The result of the derivation is a regular tree grammar G_{pt} with, $\Sigma = \{$ ~, &, |, (,), *true*, *false*, ϵ, $(\mathrm{B},1)$, $(\mathrm{B},2)$, $(\mathrm{B},3)$ $\}$, nonterminals $N = \{\mathrm{B}\}$, start symbol $S = \mathrm{B}$ and rules

$$\begin{array}{l}
\mathit{Prods} = \{ \\
\quad \mathrm{B} \rightarrow (\mathrm{B},2)(\text{“}\sim\text{”},\mathrm{B}), \\
\quad \mathrm{B} \rightarrow (\mathrm{B},3)(\mathrm{B},\text{“\&”},\mathrm{B}), \\
\quad \mathrm{B} \rightarrow (\mathrm{B},3)(\mathrm{B},\text{“}\,|\,\text{”},\mathrm{B}), \\
\quad \mathrm{B} \rightarrow (\mathrm{B},3)(\text{“(”},\mathrm{B},\text{“)”}), \\
\quad \mathrm{B} \rightarrow (\mathrm{B},1)(\text{“}\mathit{true}\text{”}), \\
\quad \mathrm{B} \rightarrow (\mathrm{B},1)(\text{“}\mathit{false}\text{)”} \\
\}
\end{array}$$

The following statements hold for context-free grammars and regular tree grammars:

- ◇ $\mathcal{L}(G_{pt}) = \mathit{parse}(\mathcal{L}(G_{cfg}))$
- ◇ $\mathcal{L}(G_{cfg}) = \mathit{yield}(\mathcal{L}(G_{pt}))$. Hence, also
- ◇ $\mathcal{L}(G_{cfg}) = \mathit{yield}(\mathit{parse}(\mathcal{L}(G_{cfg})))$
- ◇ $\mathcal{L}(G_{pt}) = \mathit{parse}(\mathit{yield}(\mathcal{L}(G_{pt})))$

Sentences can be mapped to a parse tree and back to the original sentence. This is a result of the fact that the *parse* function does not throw away information, but it builds a tree with the original sentence distributed over its leaves.

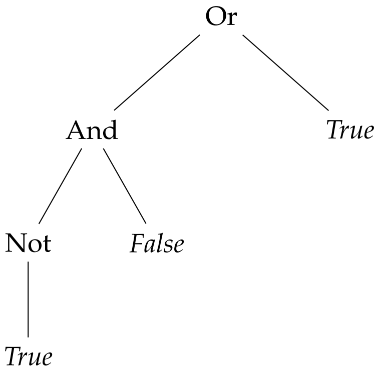
2.5 Abstract Syntax Trees

The set of abstract syntax trees of a language is called the abstract syntax and is defined by a regular tree grammar. These abstract syntax trees are considered as the abstract representation of well-formed sentences [36]. The abstract syntax representation of a sentence is unique, while the textual representation is usually cluttered with optional and semantically irrelevant details such as blanks and line feeds. These optional and semantically irrelevant details do we call *syntactic sugar*. On the other hand the *semantics* of a language is concerned with the meaning of grammatically correct programs [87].

The *abstract syntax tree* is a representation of a sentence without superfluous nodes, such as nodes corresponding to keywords and *chain rules* [76]. A chain rule is a grammar rule of the form $n_1 \rightarrow n_2$, where both n_1 and n_2 are nonterminals.

Example 2.5.1. (Abstract syntax tree). An example of an abstract syntax tree of the sentence s given in Example 2.2.3 is:

$AST(s) =$



It can also be represented as a term

$$AST(s) = Or(And(Not(True), False), True).$$

2.6 Used Languages and Formalisms

Throughout this thesis we use a number of formalisms. This section discusses these formalisms, provides their syntax and relates it to the presented theory. Furthermore, the language PICO, used for illustrating purposes, is also presented here.

2.6.1 The PICO Language

The goal of PICO [13] is to have a simple language, large enough to illustrate the concepts of parsing, type checking and evaluation. First we give a small introduction of the PICO language and its grammar.

Informal, the PICO language is the language of while-programs. The main features of PICO are:

- Two types: natural numbers and strings.
- Variables must be declared in a separate section.
- Expressions can be made of constants, variables, addition, subtraction and concatenation.
- Statements: assignment, if-then-else and while-do.

A PICO program consists of declarations followed by statements. Variables must be declared before they can be used in the program. Statements and

```
begin declare input : natural,
              output : natural,
              repnr : natural,
              rep : natural;
      input := 14;
      output := 1;
      while input - 1 do
          rep := output;
          repnr := input;
          while repnr - 1 do
             output := output + rep;
             repnr := repnr - 1
          od;
          input := input - 1
      od
end
```

Figure 2.1 A PICO program.

expressions can be used in the body of the program . An example PICO program that computes the factorial function is given in Figure 2.1[2].

2.6.2 Syntax Definition Formalism

Template grammars, as we will present in Chapter 5, can easily become very ambiguous and dealing with ambiguities is a primary requirement for parsing templates. The Scannerless Generalized LR (SGLR) algorithm, and its implementation the SGLR parser [117], can deal with these ambiguities out of the box. The SGLR parser comes with the Syntax Definition Formalism (SDF) [55] for defining grammars, which is the main reason for using SDF in this thesis.

In contrast with other parser algorithms, such as LL or LALR, and their used BNF-like [9] grammar formalisms, SDF supports the complete class of context-free grammars. This results in the ability of SDF to support modular grammar definitions. Pieces of grammar can be embedded in modules and imported by other modules. While combining grammars it is even possible to parameterize generic modules with nonterminals. The modularity enables combining and reusing of grammars.

The core of an SDF module consists of the elements of the mathematically four-tuple definition of a context-free grammar as defined in Section 2.2. In SDF nonterminals are called *sorts* and declared after the similar keyword `sorts`. *Symbols* is the global name for literals, sorts and character classes and form the elementary building blocks of SDF syntax rules. Start symbols are declared after the keyword `context-free start-symbols`. Production rules are declared

[2] Example borrowed from http://www.meta-environment.org/doc/books//extraction-transformation/language-definitions/language-definitions.html (accessed on November 30, 2010)

```
module languages/pico/syntax/Pico

imports basic/NatCon
imports basic/StrCon
imports basic/Whitespace

hiddens
  context-free start-symbols
    PROGRAM

exports
  sorts PROGRAM DECLS ID-TYPE STATEMENT EXP TYPE PICO-ID

  context-free syntax
    "begin" DECLS {STATEMENT";" }* "end"
                          -> PROGRAM {cons("program")}
    "declare" {ID-TYPE "," }* ";"
                          -> DECLS {cons("decls")}
    PICO-ID ":" TYPE      -> ID-TYPE {cons("decl")}

    PICO-ID ":=" EXP      -> STATEMENT {cons("assignment")}
    "if" EXP "then" {STATEMENT ";" }*
             "else" {STATEMENT ";" }* "fi"
                          -> STATEMENT {cons("if")}
    "while" EXP "do" {STATEMENT ";" }* "od"
                          -> STATEMENT {cons("while")}

    PICO-ID               -> EXP {cons("id")}
    NatCon                -> EXP {cons("natcon")}
    StrCon                -> EXP {cons("strcon")}
    EXP "+" EXP           -> EXP {cons("add")}
    EXP "-" EXP           -> EXP {cons("sub")}
    EXP "||" EXP          -> EXP {cons("concat")}
    "(" EXP ")"           -> EXP {cons("bracket")}

    "natural"             -> TYPE {cons("natural")}
    "string"              -> TYPE {cons("string")}

  lexical syntax
    [a-z] [a-z0-9]*       -> PICO-ID  {cons("picoid")}

  lexical restrictions
    PICO-ID -/- [a-z0-9]
```

Figure 2.2 The PICO grammar in SDF.

in sections `context-free syntax` and `lexical syntax`. The *productions rules* contain a *syntactical pattern* at the left-hand side and a resulting sort at the right-hand side. This left-hand side pattern is based on a combination of symbols, i.e. terminals in combination with nonterminals. Symbols can be declared as optional via a postfix question mark. In the `context-free syntax` section a `Layout` sort is automatically injected between every symbol in the left-hand side of a production rule. The `Layout` sort is an SDF/SGLR embedded sort for white spaces and line feeds. To illustrate SDF, the PICO language is defined by the SDF module shown in Figure 2.2.

SDF also supports concise declaration of associative lists. A list is declared by its elements and a postfix operator * or +, with the respectively meaning of at least zero times or at least one time. Lists may also contain a separator, which are declared via the pattern `{Symbol Literal}*`, where `Symbol` defines the syntax of the elements and `Literal` defines the separator syntax, for example: `{STATEMENT ";" }*`.

The production rules can be annotated with a list of properties between curling brackets at the right-hand side of the rule. The parser includes these annotations in the parse tree at the node produced by the production rule. Tools processing the parse tree can use these annotations.

For example, an abstract syntax for an SDF grammar can be specified using annotations. Every production rule can be annotated with a constructor value, which is used to instantiate a node in the abstract syntax tree during desugaring. The constructor is declared via a `cons` value. SDF requires that a constructor is unique for a given sort, and in that way suffices the first requirement of our *desugar* function of Definition 3.2.2. It does not require that a constructor is only used for a fixed rank and thus SDF does not satisfy the requirements to generate a legal regular tree language. Production rules can also annotated with the keyword `reject`. The `reject` annotation specifies that strings specified by the rule are rejected for that nonterminal.

Besides these core features of SDF, it supports modularization of grammar definitions. Every grammar definition file is declared as a module with a name, which can be imported by other modules. Modules are imported via the `imports` keyword followed by the name(s) of imported modules. Sections of a grammar module can be declared hidden or visible via the keywords `hiddens` and `exports` to prevent unexpected collisions between grammar modules result in undesired ambiguities. Exported sections are visible in the entire grammar, while hidden sections are only visible in the local grammar module. SDF also provides syntax to define priorities and associativity to express disambiguation rules in a grammar. For further information about SDF, we refer to the SDF documentation [55].

Considering again the SDF module shown in Figure 2.2. The nonterminals and production rules are declared in the exported section. The start symbol is `PROGRAM`, which is the root sort for a PICO program. In the PICO module the start symbol is declared hidden to prevent automatic propagation to modules importing this grammar. The annotation feature of SDF is also used in the PICO module to specify the abstract syntax tree. The definition for the sorts `NatCon` and `StrCon`, and a module defining white space (spaces, tabs, and new lines) are imported.

The grammar of Figure 2.2 is used to parse PICO programs, like Figure 2.1. The abstract syntax tree, result of parsing the program and desugaring the parse tree is shown in Figure 2.3. The tree is displayed in the ATerm format, discussed in Section 2.6.3.

```
program(
  decls([
    decl( "input", natural ),
    decl( "output", natural ),
    decl( "repnr", natural ),
    decl( "rep", natural )
  ]),
  [
    assignment( "input", natcon( 14 ) ),
    assignment( "output", natcon( 1 ) ),
    while( sub( id( "input" ), natcon( 1 ) ),[
      assignment( "rep", id( "output" ) ),
      assignment( "repnr", id( "input" ) ),
      while( sub( id( "repnr" ), natcon ( 1 ) ),[
        assignment( "output", add( id( "output" ),
                                   id( "rep" ) ) ),
        assignment( "repnr", sub( id( "repnr" ),
                                  natcon( 1 ) ) )
      ]),
      assignment( "input", sub( id( "input" ),
                                natcon( 1 ) ) )
    ])
  ]
)
```

Figure 2.3 Abstract syntax tree of PICO program of Figure 2.1.

2.6.3 ATerms

The syntax for terms used in this thesis is based on a subset of the ATerms syntax [22]. The ATerms syntax is supported by an implementation for Java and C, which we use in our prototype(s) and case studies.

ATerms have support for lists, which are not directly supported by the presented regular tree grammars. In our approach lists must be binary trees to stay fully compatible with the regular tree grammars. The serialized term notation of the list is only a shorthand notation for these binary trees, i.e. the list

```
[ "a", "b", "c" ]
```

has the internal representation

```
[ "a", [ "b", [ "c" , []]]].
```

Since the lists of ATerms are internally stored as binary trees, where the left branch is the element and right branch the list or empty list, the use of ATerms meets this requirement. The subset of the ATerm language is defined by the SDF definition of Figure 2.4.

The `IdCon` and `StrCon` are respectively defined as the following character classes `[A-Za-z][A-Za-z\-]*`[3] and `["]~[\0-\31\n\t\"\\]*["]`.

[3] The original character class for IdCon allows numeric symbols in the tail. We do not allow them to prevent ambiguities in our tree path queries.

```
module ATerms

imports StrCon
        IdCon

exports
  sorts AFun ATerm

  context-free syntax
        StrCon  -> AFun
        IdCon   -> AFun
        AFun                               -> ATerm
        AFun "(" {ATerm ","}+ ")"          -> ATerm
        "[" {ATerm ","}* "]"               -> ATerm
```

Figure 2.4 Subset of ATerm syntax used in this thesis.

3
The Unparser

The translators of a regular tree to a sentence of a context-free language belong to the class of meta programs for instantiating code. A basic but special variant of this kind of meta programs is the unparser. The unparser translates an abstract syntax tree into a sentence of a context-free language. The unparser is semantically neutral, that means, the abstract syntax tree can be reproduced by parsing and desugaring the output sentence of an unparser. The unparser is capable to instantiate all sentences of the output language, modulo layout and other semantically irrelevant syntax. Therefore a metalanguage must be able to express unparsers, so it is not a limiting factor for writing meta programs for generating code. This chapter discusses the properties and requirements of the unparser and shows that a linear deterministic tree-to-string transducer is powerful enough to express the unparser.

3.1 Introduction

The goal of implementing meta programs for code instantiating is to translate an input programming language to an output programming language. Since the scope of these programs is limited to computer languages, we use formal language theory to obtain the requirements for this kind of meta programs. We discuss the relations between concrete syntax, parse tree, abstract syntax, parser, unparser and their underlying grammars.

An overview of these relations is given in Figure 3.1. The circles are sets of sentences or trees defined by the languages discussed in Chapter 2. The boxes represent the mapping functions between these sets. These relations are used to define the requirements for the metalanguage. In Section 3.2, we start with the discussion of the *desugar* function, which maps an abstract syntax tree to a parse tree . This function is called *desugar*, since all superfluous syntactical information is removed. In Section 3.3 the *unparse* function is

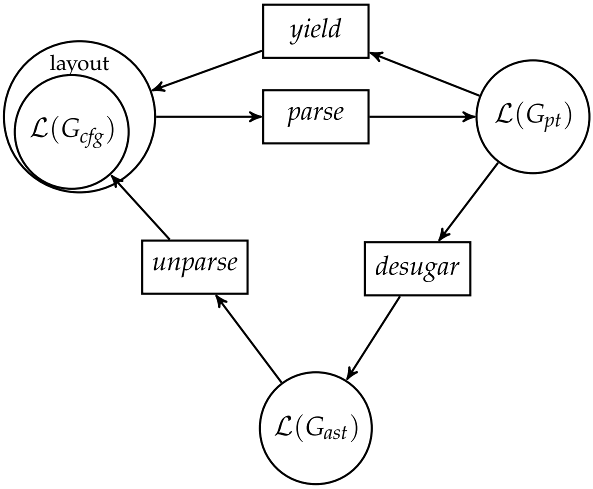


Figure 3.1 Relations between languages and their grammars.

presented, which is used to obtain a concrete syntax from an abstract syntax tree. After that, unparser-completeness is discussed in Section 3.4 and we conclude with conclusions in Section 3.5.

3.2 Deriving Abstract Syntax Trees

This section discusses the transformation of a parse tree to an abstract syntax tree (see Section 2.5). The requirement for this transformation is that the *meaning* of the original sentence is not altered during the translation from a parse tree to an abstract syntax tree. This requirement is visualized in Figure 3.1 by the cycle of the mappings *unparse*, *parse* and *desugar*. We define the meaning of a program by the following requirement: Given a parse tree t_{pt} and an abstract syntax tree t_{ast} representing the same piece of object code and given a function f_1 operating on t_{pt}, resulting in a term t, and f_2 operating on t_{ast} resulting in the same t, it is required that $f_1(t_{pt}) = f_2(desugar(t_{pt}))$. For example, the parse tree of a C program and the abstract syntax tree of the same program should result in the same assembler code after compilation. By means of that definition, the required detail of the abstract syntax is dependent on the information used by the function f_2 processing it. In other words, a function only interested in a subset of the semantics of a programming language can use a less detailed abstract syntax than a function using every detail of the programming language.

The topic of this thesis is not to design the most compact abstract syntax for a given f_2, but we need a formal notion of abstract syntax to discuss the properties of meta programs instantiating code. In practice, an abstract syntax tree is based on the parse tree modulo layout information and keywords. We will provide in the next paragraphs an approach for transforming a parse tree to an abstract syntax tree.

The *desugar* function can be manually defined in the parser definition, like in parser implementations such as YACC [65], ANTLR [93] and Beaver[1]. These parsers allow to associate a production rule with a *semantic action* in the grammar. These semantic actions, for example specified in a third generation programming language, can directly instantiate an abstract syntax tree.

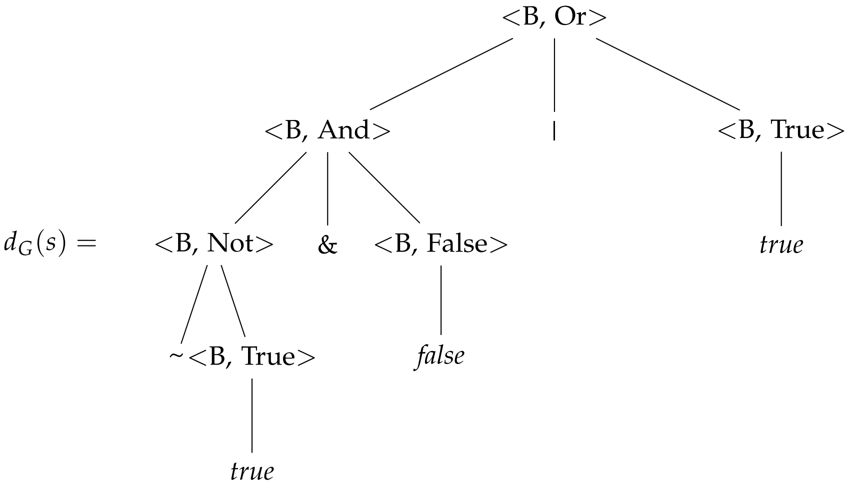
In case of a parser based solely on a context-free grammar without semantic actions, such as SGLR [117], the output is a parse tree. An intermediate step is necessary to transform this parse tree to an abstract syntax tree. An approach based on a set of heuristics can be used to transform such a parse tree to an abstract syntax tree, for example presented by Wile [124]. However, this approach will introduce machine generated names for the nodes. In order to give full control of the node labels in the abstract syntax, we present a solution based on augmenting the production rules of a context-free grammar with a signature label to express this transformation scheme. This signature label is used to construct a node in the abstract syntax tree. A production rule in the augmented context-free grammar has the form $n \rightarrow y\{c\}$ where $n \in N$, $y \in (N \bigcup \Sigma)*$ and c is an element of an alphabet Σ_c, not necessarily disjoint from $N \bigcup \Sigma$. Σ_c is the alphabet of the regular tree grammar belonging to the abstract syntax. The signature label c is not necessarily equal to the nonterminal of the production rule, but it is required that a signature label c is utmost used once for a nonterminal. In other words the pair (n, c), where $n \in N$ and $c \in \Sigma_c$, must be unique (see Definition 3.3.2). The signature label is stored as a tuple in the node labels of the parse tree to use it for the transformation.

Example 3.2.1. (Augmented context-free grammar) The following set of production rules show the extension of the context-free grammar of Example 2.2.2 with signature labels:

[1] http://beaver.sourceforge.net/ (accessed on November 30, 2010)

$$
\begin{aligned}
&\mathit{Prods} = \{\\
&\quad \mathrm{B} \rightarrow \sim \mathrm{B}\{\mathit{Not}\}\\
&\quad \mathrm{B} \rightarrow \mathrm{B}\ \&\ \mathrm{B}\{\mathit{And}\}\\
&\quad \mathrm{B} \rightarrow \mathrm{B} \mid \mathrm{B}\{\mathit{Or}\}\\
&\quad \mathrm{B} \rightarrow (\ \mathrm{B}\ )\{\mathit{Br}\}\\
&\quad \mathrm{B} \rightarrow \mathit{true}\{\mathit{True}\}\\
&\quad \mathrm{B} \rightarrow \mathit{false}\{\mathit{False}\}\\
&\}
\end{aligned}
$$

The parse tree of s =`~true & false | true` using the grammar with signature information will have the following form:



and the term representation is:

$d_G(s) =$

```
< B , Or >(
   < B, And >(
        < B, Not >( ~, <B, True >( true )),
        &,
        < B, False >( false )
    ),
   |,
   < B, True >( true )
)
```

Having a context-free grammar with signature labels, the abstract syntax tree can be automatically instantiated from a parse tree. This is executed by the *desugar* function, which consists of two transformations. First it removes *chain production rules*, i.e. rules of the form $n_1 \rightarrow n_2$ in the context-free grammar. Second it replaces the nodes in a parse tree with new nodes, which are labeled by a *signature label* c. The new nodes only contain children for the nonterminals of the original nodes. As a result, the rank of signature label c is equal to the number of nonterminals in the corresponding production rule of the context-free grammar.

Definition 3.2.2. (Desugar). The *desugar* function is defined by the following equations:

- ◇ $desugarchildren() = \epsilon$;
- ◇ $desugarchildren(a, x_1, \ldots, x_k) = desugarchildren(x_1, \ldots, x_k)$, where $a \in \Sigma$;
- ◇ $desugarchildren(x_1, x_2, \ldots, x_k) = desugar(x_1), desugarchildren(x_2, \ldots, x_k)$, where $x_1 \in N$;
- ◇ $desugar(< f, c > (x)) = desugar(x)$, where $x \in N$ (this equation removes chain rules);
- ◇ $desugar(< f, c > (x_1, \ldots, x_k)) = c$, when $desugarchildren(x_1, \ldots, x_k) = \epsilon$;
- ◇ $desugar(< f, c > (x_1, x_2, \ldots, x_k)) = c(desugarchildren(x_1, x_2, \ldots, x_k))$, when $desugarchildren(x_1, x_2, \ldots, x_k) \neq \epsilon$.

Applying the *desugar* function to the parse tree of Example 3.2.1 will result in the abstract syntax tree of Example 2.5.1.

Theorem 3.2.3. The abstract syntax tree obtained by applying the *desugar* function to a parse tree belongs to a regular tree language [32].

(Proof) Recognizability of trees by finite tree automata is closed under linear tree homomorphism [39]. The *desugar* function is a linear tree homomorphism; subtrees are only removed and not duplicated. Since the abstract syntax tree is a linear tree homomorphism of the parse tree and the set of parse trees of a context-free language is a regular tree language [33], the abstract syntax tree belongs to a regular tree language. □

The mapping of a parse tree to an abstract syntax tree is regular, which suggests there exists a mapping between the regular tree grammar of the parse tree G_{pt} and the regular tree grammar of the abstract syntax tree G_{ast}. We show that there is indeed a mapping:

Definition 3.2.4. (Mapping parse tree grammar to abstract syntax tree grammar). Let $G_{pt} = \langle \Sigma, N, S, Prods \rangle$ be the regular tree grammar of the parse tree, the regular tree grammar $G_{ast} = \langle \Sigma', N', S', Prods' \rangle$ is derived via the following rules:

- ◇ The start symbol of both grammars is equal: $S = S'$,
- ◇ The set of nonterminals of both grammars is equal: $N = N'$,
- ◇ The alphabet of G_{ast} is: $\Sigma' = \Sigma_c$. Σ_c is the alphabet containing the labels $c_1, \ldots, c_p$ used for augmenting the context-free grammar with signature labels.
- ◇ The set of productions $Prods'$ of G_{ast} is derived by the following rules:

if $< A, c > (z_1 z_2 \ldots z_k) \in Prods$ then:

$A \rightarrow c(getnonterminals(z_1, z_2, \ldots, z_k)) \in Prods'$,
when $getnonterminals(z_1, z_2, \ldots, z_k) \neq \epsilon$;

$A \rightarrow c \in Prods'$,
when $getnonterminals(z_1, \ldots, z_k) = \epsilon$;

if $< A, c > (z) \in Prods$ then:

$A \rightarrow z \in Prods'$, when $z \in N$;

$getnonterminals(z_1, z_2 \ldots, z_j) = getnonterminals(z_2, \ldots, z_j)$,
when $z_1 \in \Sigma$;

$getnonterminals(z_1, z_2 \ldots, z_j) = z_1, getnonterminals(z_2, \ldots, z_j)$,
when $z_1 \in N$.

$getnonterminals() = \epsilon$.

Example 3.2.5. (Mapping G_{pt} to G_{ast}). The result of the mapping of G_{pt} to G_{ast} for the language of Example 2.2.2 is a regular tree grammar with $\Sigma = \{$ And(,), Or(,), Not(), Br(), True, False $\}$, nonterminals $N = \{\text{B}\}$, start symbol $S = \text{B}$ and rules

$$
\begin{aligned}
Prods = \{ & \\
& \text{B} \rightarrow \text{Not}(\text{B}), \\
& \text{B} \rightarrow \text{And}(\text{B}, \text{B}), \\
& \text{B} \rightarrow \text{Or}(\text{B}, \text{B}), \\
& \text{B} \rightarrow \text{Br}(\text{B}), \\
& \text{B} \rightarrow \text{True}, \\
& \text{B} \rightarrow \text{False} \\
\} &
\end{aligned}
$$

For the sake of completeness, the next theorem shows that the mapping G_{pt} to G_{ast} of Definition 3.2.4 indeed holds for every abstract syntax tree resulting of desugaring $t_{pt} \in \mathcal{L}(G_{pt})$.

Theorem 3.2.6. Given a G_{pt} augmented with the signature label, there is a regular tree grammar G_{ast} defining the set of abstract syntax trees resulting of desugaring the parse trees of G_{pt}.

(proof) We show for each form of production rule in the Chomsky normal form (see Section 2.2), that the mapping of Definition 3.2.4 results in a G_{ast} producing the languages of desugaring $t_{pt} \in \mathcal{L}(G_{pt})$.

The first production rule in G_{cfg} is $A \rightarrow \epsilon\{c_1\}$. Following Definition 2.4.1, the production rule corresponding to G_{pt} is $A \rightarrow < A, c_1 > (\epsilon)$[2]. Mapping this rule to the G_{ast} grammar results in the rule $A \rightarrow c_1$. Parsing the empty string produces the parse tree: $parse(\epsilon) =< A, c_1 > (\epsilon)$. Desugaring this parse tree produces the abstract syntax tree: $desugar(< A(\epsilon), c_1 > (\epsilon)) = c_1$. The abstract syntax tree c_1 is the only tree of $\mathcal{L}(G_{ast})$.

The second production rule in G_{cfg} is $A \rightarrow s\{c_2\}$. Following Definition 2.4.1, the rule belonging to G_{pt} is $A \rightarrow < A, c_2 > (s)$. Mapping this rule to the G_{ast} grammar results in the rule $A \rightarrow c_2$. Parsing the string s produces the parse tree: $parse(s) =< A, c_2 > (s)$. Desugaring this parse tree produces the abstract syntax tree: $desugar(< A, c_2 > (s)) = c_2$. The abstract syntax tree c is the only tree of $\mathcal{L}(G_{ast})$.

Finally, we need the inductive case based on the production rule $A \rightarrow n_1 n_2\{c_3\}$. Given a G_{cfg} with start symbol A and the production rules

$$\begin{aligned} A &\rightarrow s\{c_2\} \\ A &\rightarrow AA'\{c_3\} \\ A' &\rightarrow s\{c_4\} \end{aligned}$$

Following Definition 2.4.1, the rules belonging to G_{pt} are $A \rightarrow < A, c_2 > (s)$, $A \rightarrow < A, c_3 > (A, A')$ and $A' \rightarrow < A', c_4 > (s)$. Mapping this rule to the G_{ast} grammar results in the production rule $A \rightarrow c_2$, $A \rightarrow c_3(A, A')$ and $A' \rightarrow c_4$. Given a string $s \cdot s$, parsing this string produces the parse tree: $parse(s \cdot s) = < A, c_3 > (< A, c_2 > (s), < A, c_4 > (s))$. Desugaring this parse tree produces the abstract syntax tree:

$$desugar(< A, c_3 > (< A, c_2 > (s), < A, c_4 > (s))) = c_3(c_2, c_4).$$

The abstract syntax tree $c_3(c_2, c_4)$ is a tree of $\mathcal{L}(G_{ast})$.

□

[2] Note: we omitted the transformation of the symbol A into (A, k) for readability reasons.

3.3 The Unparser

We have presented the relations between concrete syntax, its abstract syntax and their grammars. The original sentence can be reconstructed from a parse tree using the *yield* function, but in case of an abstract syntax tree this *yield* function cannot be used.

In contrast with the *parse/yield* couple, which can always restore the original code including layout from a parse tree, this is not the case for abstract syntax trees. The abstract syntax trees lack information to reconstruct the original sentence, since the original keywords are implicitly stored in the nodes and not explicitly in leaves. Per G_{ast} a function is necessary to reconstruct a concrete syntax representation of its abstract syntax trees. This function is called the *unparse* function: It translates an abstract syntax tree into a textual representation of the sentence [24].

In contrast with the *yield* function, it is not possible to guarantee that unparsed code is syntactically equivalent to the parsed code. Superfluous information, like layout, is not present in the abstract syntax tree and has to be induced by default rules in the unparser definition. The *unparse* function can indeed be used as a code formatter [24]. The unparser cannot restore the original code for an arbitrary case, except the one, where the layout of the original code matches the layout syntax defined in the unparser. The following relation reflects this property:

$$\mathcal{L}(G_{cfg}) \supseteq unparse(desugar(parse(\mathcal{L}(G_{cfg}))))$$

However, the *unparse* function should produce a text which is syntactically correct and represents the original abstract syntax tree. The unparser is correct if and only if re-parsing its output sentences reproduce the same abstract syntax trees as the original inputs [97]. That is, as the couple of *unparse* and *desugar* only executes a syntactical mapping and does not alter the meaning of the code represented by the concrete syntax or the abstract syntax. The abstract syntax contains all semantic information, and this information should be present in the unparsed sentence without altering it. Parsing and desugaring this unparsed code must result in the same abstract syntax tree, otherwise information is lost or altered somewhere in the process. The following relation reflects this property:

$$\mathcal{L}(G_{ast}) = desugar(parse(unparse(\mathcal{L}(G_{ast}))))$$

The signature of the*unparse* function is:

$$unparse : Tree \rightarrow String$$

This signature reflects the property that the input of the unparser is a tree and the result a string. The *unparse* function traverses a tree, just as the *yield* function does. The unparser differs from the *yield* function (see Definition 2.2.4) as it is not a generic tree traversal function, but is tailored for the abstract syntax grammar of the input. The *yield* function is a traversal function walking every tree structure till it detects a leaf, while an *unparse* function has an action for every production rule in the regular tree grammar of the abstract syntax. The *unparse* function restores the mapping defined by the *desugar* function. Where the *desugar* function removes terminals from the parse tree, the unparser actions must restore the removed terminals. Terminals are removed from nodes at different levels in the tree and to restore them actions must be defined for the nodes where these terminals must be restored. As a result the *unparse* function follows the structure of the abstract syntax grammar and can only be applied to trees produced by that grammar.

The *unparse* function can be derived from the context-free grammar of the output language, extended with signature information. For each production rule in the context-free grammar (minus the chain rules) an action is defined in the *unparse* function to traverse the abstract syntax tree and restore terminals whenever it is needed. An action of the *unparse* function has a left-hand side and a right-hand side.

The left-hand side *matches* on a node in the abstract syntax tree with the signature label c and the variables $x_1, \ldots, x_k$. The rank of c is not necessarily equal to k since the rank of c is equal to the number of nonterminals in the pattern of the production rule belonging to c. Therefore the variable x_i only exists if a symbol is a nonterminal at index i in the right-hand side of the corresponding production rule.

The right-hand side of the action is a copy of the pattern described by the corresponding production rule. It constructs a string $s_1 \cdot \ldots \cdot s_k$, where $s_1, \ldots, s_k$ are strings or calls to the *unparse* function and $\cdot$ denotes the concatenation operation. The number of strings k in the right-hand side is equal to the number of symbols in the pattern of the corresponding production rule. At the index i, when a terminal is defined in the production rule, the string s_i is equal to the terminal symbol. In case of a nonterminal, s_i is a call to the unparser with the variable x_i as argument.

For example, consider the context-free production rule $\mathrm{B} \to \mathrm{B}$ & $\mathrm{B}\{And\}$. The node for this production rule in the abstract syntax tree has the following pattern $\mathrm{B} \to \mathrm{And}(\mathrm{B}, \mathrm{B})$. The left-hand side of the unparse action will become $unparseB(\mathrm{And}(x_1, x_3))$. x_2 is not available in the left-hand side, since the second symbol in the context-free production rule is a terminal "&". The right-hand side of the unparse action is $unparseB(x_1) \cdot$ "&" $\cdot\ unparseB(x_3)$.

The actions of the *unparse* function are specific for a nonterminal in the regular tree grammar of the abstract syntax. The function name of the left-hand side of the action has a postfix to indicate for which nonterminal the action applies, for example B. The postfix of the called *unparse* functions in the right-hand side of the action is equal to the nonterminal where x_i corresponds with in the context-free production rule.

An unparser is derived from a context-free grammar by applying the following rules:

Definition 3.3.1. (Unparse). The *unparse* function contains a set of actions *Actions*, which are derived from the context-free grammar augmented with signature labels $G_{cfg} = \langle \Sigma, N, S, Prods \rangle$. The set of actions is obtained by the following rule:

- ◇ if $A \to z_1 \ldots z_k\{c\} \in Prods$ then:
  $$unparseA(c(makeLhs(z_1, \ldots, z_k, 1))) = makeRhs(z_1, \ldots, z_k, 1) \in Actions, \text{ when } makeLhs(z_1, \ldots, z_k, 1) \neq \epsilon;$$
  $$unparseA(c) = makeRhs(z_1, \ldots, z_k, 1) \in Actions, \text{ when } makeLhs(z_1, \ldots, z_k, 1) = \epsilon;$$
- ◇ $makeLhs(z_1, z_2 \ldots, z_j, i) = makeLhs(z_2, \ldots, z_j, i+1)$, when $z_1 \in \Sigma$;
- ◇ $makeLhs(z_1, z_2 \ldots, z_j, i) = x_i, makeLhs(z_2, \ldots, z_j, i+1)$, when $z_1 \in N$;
- ◇ $makeLhs(z_1, i) = \epsilon$, when $z_1 \in \Sigma$;
- ◇ $makeLhs(z_1, i) = x_i$, when $z_1 \in N$;
- ◇ $makeLhs() = \epsilon$;
- ◇ $makeRhs(z_1, z_2 \ldots, z_j, i) = z_1 \cdot makeRhs(z_2, \ldots, z_j, i+1)$, when $z_1 \in \Sigma$;
- ◇ $makeRhs(z_1, z_2 \ldots, z_j, i) = unparseZ_1(x_i) \cdot makeRhs(z_2, \ldots, z_j, i+1)$, when $z_1 \in N$;
- ◇ $makeRhs(z_1, i) = z_1$, when $z_1 \in \Sigma$;
- ◇ $makeRhs(z_1, i) = \epsilon$, when $z_1 \in N$;
- ◇ $makeRhs() = \epsilon$.

Note that x_i in the the right-hand side of *makeLhs* and *makeRhs* are only bound in the instantiated *unparse* function, see Example 3.3.3.

The *desugar* function removes nodes in the parse tree belonging to chain rules in the context-free grammar. An unparser based on a context-free grammar with chain rules contains for these chain rules an action of the form: $unparseA_1(c(x)) = unparseA_2(x)$. But the abstract syntax tree generated by the *desugar* function never contains a node of the form $c(t)$, so the mapping of A_1 to A_2 is never triggered. Following the process of the *desugar* function, we could generate for chain rules an unparse rule of the form $unparseA_1(x) = unparseA_2(x)$, but this rule introduces potential nondeterministic behavior, since the left-hand side matches all terms and does not filter it on the label of the root node of the term. Therefore, we require that the context-free grammar used to derive an unparser does not contain chain rules. In case a context-free grammar contains chain rules of the form $A_1 \rightarrow A_2$, these rules are removed from the grammar and all occurrences of A_2 in the grammar are replaced by A_1.

Theorem 3.3.2. An unparser is linear and deterministic.

(proof) An unparser is linear if for each action no x_i occurs more than once in its right-hand side. Consider the actions of Definition 3.3.1; the right hand side of an unparser action never contains a duplication of a variable.

Observe that if it contains a duplicated meta-variable, it will break the requirement of the unparser that it should not alter the meaning of the abstract syntax tree. The relations

$$\mathcal{L}(G_{ast}) = desugar(parse(unparse(\mathcal{L}(G_{ast}))))$$
$$\mathcal{L}(G_{cfg}) \supseteq unparse(desugar(parse(\mathcal{L}(G_{cfg}))))$$

will not hold anymore. For example, in case a variable is duplicated

$$unparseB(And(x_1, x_3)) = unparseB(x_1) \cdot \text{“\&”} \cdot unparseB(x_1).$$

The relations will only hold for abstract syntax trees of the form $And(x_1, x_3)$, where x_1 and x_3 are (recursively) equal. In other situations, parsing and desugaring the output results in another abstract syntax tree than used for the input.

The unparser is deterministic if the different actions have different left-hand sides. This requirement holds for the derived unparser, since it is a requirement for the augmented context-free tree grammar that the tuples (n, c), where n is the left-hand side nonterminal of a production rule and c the signature label of that production rule, are unique. A nonterminal directly corresponds with an action *unparseA* in the unparser, and the signature label corresponds with the label used in the left-hand side of the unparser actions $unparseA(c(x_1, \ldots, x_k))$. Since the tuples (n, c) are unique, the left-hand sides of the unparser are unique and thus the unparser is deterministic. □

Example 3.3.3. (Unparse). The *unparse* function to unparse the abstract syntax tree of Example 2.5.1 is obtained by applying the rules of Definition 3.3.1 to the context-free grammar of Example 3.2.1:

$$\begin{aligned}
\mathit{unparseB}(\mathit{Not}(x_2)) &= \text{“\textasciitilde”} \cdot \mathit{unparseB}(x_2)\\
\mathit{unparseB}(\mathit{And}(x_1, x_3)) &= \mathit{unparseB}(x_1) \cdot \text{“\&”} \cdot \mathit{unparseB}(x_3)\\
\mathit{unparseB}(\mathit{Or}(x_1, x_3)) &= \mathit{unparseB}(x_1) \cdot \text{“ | ”} \cdot \mathit{unparseB}(x_3)\\
\mathit{unparseB}(\mathit{Br}(x_2)) &= \text{“(”} \cdot \mathit{unparseB}(x_2) \cdot \text{“)”}\\
\mathit{unparseB}(\mathit{True}) &= \text{“true”}\\
\mathit{unparseB}(\mathit{False}) &= \text{“false”}
\end{aligned}$$

Applying this function to the abstract syntax tree

$$\mathit{Or}(\mathit{And}(\mathit{Not}(\mathit{True}), \mathit{False}), \mathit{True})$$

will result in

$$\mathit{unparseB}(\mathit{Or}(\mathit{And}(\mathit{Not}(\mathit{True}), \mathit{False}), \mathit{True})) = \text{\textasciitilde true\&false | true}$$

This example shows that the layout of the original sentence is lost and all spaces are removed. Parsing and desugaring the result of this *unparse* function instantiates an abstract syntax tree equal to the original abstract syntax tree. We assume that brackets are explicitly available in the abstract syntax tree to restore the priorities and associativities of operators correctly.

In this case removing the spaces has not changed the meaning of the boolean expression. For some languages it is not allowed to yield a string without a whitespace character between the unparsed sub strings, since two unparsed sub strings can become one string without a natural separation. In that case the right-hand patterns of the unparser must be $s_1 \cdot \sqcup \cdot \ldots \cdot \sqcup \cdot s_k$, where $\sqcup$ is a whitespace character. For most languages this is a space, but for some exotic languages, like whitespace[3], another character must be used.

An example of this problem is a sequence of two identifiers which are separated by a space in the original code. They will be parsed as a single identifier when they are concatenated by an unparser without using a whitespace character separating them. This can happen in a language such as Java where method declarations have a type followed by a method name, where both can be an identifier.

3.4 Unparser Completeness

The unparser has two specific properties (see Section 3.2) not generally applying to other meta programs instantiating code:

[3] http://compsoc.dur.ac.uk/whitespace/ (accessed on September 23, 2010)

- ◇ An unparser should not alter the semantic meaning;
- ◇ An unparser can instantiate all meaningful sentences of the output language.

We call a metalanguage capable to express unparsers *unparser*-complete. It seems not necessary that the metalanguage capable to implement the unparser is Turing-complete. This is suggested by the fact that the unparser as defined in Definition 3.3.1 shows a lot of similarity with deterministic top-down tree-to-string transducers [40]. The coming theorems show that indeed the unparser can be expressed by a deterministic top-down tree-to-string transducer.

Definition 3.4.1. (Top-down tree-to-string transducer) [40]. A top-down tree-to-string transducer is a 5-tuple $M = \langle Q, \Sigma, \Sigma', q_0, R \rangle$, where Q is a finite set of states, Σ is the ranked input alphabet, Σ' is the output alphabet, $q_0 \in Q$ is the initial state, and R is a finite set of rules of the form:

$$q(\sigma(x_1, \ldots, x_k)) \rightarrow s_1 q_1(x_{i_1}) s_2 q_2(x_{i_2}) \ldots s_p q_p(x_{i_p}) s_{p+1}$$

with $k, p \geq 0$; $q, q_1, \ldots, q_p \in Q$; $\sigma \in \Sigma_k$; $s_1, \ldots, s_{p+1} \in \Sigma'^*$, and $1 \leq i_j \leq k$ for $1 \leq j \leq p$ (if $k = 0$ then the left-hand side is $q(c)$). M is *deterministic* if different rules in R have different left-hand sides. M is *linear* if, for each rule in R, no x_i occurs more than once in its right-hand side, i.e. no data is copied.

The class of languages a top-down tree-to-string transducer can recognize is equal to its corresponding finite tree automata [40]. In contrast with a Turing machine, a top-down tree-to-string transducer cannot change the tree on which it operates. As a result a top-down tree-to-string transducer can only accept a subset of languages which a Turing Machine can accept [33].

The next theorems will show that for each context-free grammar an unparser can be defined using a top-down tree-to-string transducer and that it is linear and deterministic. Furthermore we show that the unparsers of Definition 3.3.1 can be mapped on a top-down tree-to-string transducer. We start with the theorem that every context-free grammar can be transformed to a context-free grammar only containing productions rules of the form $A \rightarrow s_1 n_1 \ldots s_i n_i s_{i+1}$.

Lemma 3.4.2. For every context-free grammar G generating $\mathcal{L}(G)$, an equivalent context-free grammar G' exists only containing rules of the form $A \rightarrow s_1 n_1 \ldots s_i n_i s_{i+1}$, where $s_i \in \Sigma^*$ and s_i can be ϵ and $n_i \in N$.

(proof) Consider the production rules of the Chomsky normal form of Theorem 3.2.6. Production rule one and two of the Chomsky normal form are already of the form $A \rightarrow s_1 n1 \ldots s_i nis_{i+1}\{c\}$, where $i = 0$. The third rule must be translated using the following mapping $A \rightarrow n_1 n_2 = A \rightarrow s_1 n_1 s_2 n_2 s_3\{c\}$, where s_1, s_2 and s_3 are ϵ. Production rules of the form $A \rightarrow \epsilon n_1 \epsilon n_i \epsilon\{c\}$ produce the same language as rules of the form $A \rightarrow n_1 n_2\{c\}$, since ϵ is empty in

the sentences produced by the production rule. As a result a grammar based on production rules of the form $A \rightarrow s_1 n_1 \ldots s_i n_i s_{i+1}\{c\}$ produces the same language as its equivalent in the Chomsky normal form. □

Theorem 3.4.3. A unparser based on a linear deterministic top-down tree-to-string transducer can be defined for every context-free grammar augmented with signature labels.

(proof) We show that for each of three forms of production rules of the Chomsky normal form, see Section 2.2, the relation

$$\mathcal{L}(G_{ast}) = desugar(parse(unparse(\mathcal{L}(G_{ast}))))$$

holds.

First, consider the production rule $A \rightarrow \epsilon\{c_1\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_1$. The tree-to-string transducer belonging to this production rule is: $q(c_1) \rightarrow \epsilon$. Applying this tree-to-string transducer to the abstract syntax tree t_{ast} results in $q(c_1) = \epsilon$. Parsing this empty string produces the parse tree: $parse(\epsilon) =< A, c_1 > (\epsilon)$. Desugaring this parse tree produces the abstract syntax tree $desugar(< A, c_1 > (\epsilon)) = c_1$.

Second, consider the production rule $A \rightarrow s\{c_2\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_2$. The tree-to-string transducer belonging to this production rule is: $q(c_2) \rightarrow s$. Applying this tree-to-string transducer to the abstract syntax tree t_{ast} results in $q(c_2) = s$. Parsing the string s produces a parse tree $parse(s) =< A, c_2 > (s)$. Desugaring this parse tree produces the abstract syntax tree $desugar(< A, c_2 > (s)) = c_2$.

The last production rule is $A \rightarrow \epsilon n_1 \epsilon n_2 \epsilon\{c_3\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_3(t_1, t_2)$, where t_1 and t_2 are the abstract syntax trees belonging to n_1, respectively n_2. The tree-to-string transducer belonging to this production rule is: $q(c_3(x_1, x_2)) \rightarrow \epsilon q_1(x_1) \epsilon q_2(x_2) \epsilon$. Applying this tree-to-string transducer to the abstract syntax tree results in $q(c_3(t_1, t_2)) = s_1 \cdot s_2$, where $s_1 = q_1(t_1)$ and $s_2 = q_2(t_2)$, i.e. s_1 and s_2 are the result of unparsing the sub-terms t_1 and t_2. The ϵ's are omitted in the string $s_1 \cdot s_2$, since they are empty. Parsing the string $s_1 \cdot s_2$ produces a parse tree $parse(s_1 \cdot s_2) =< A, c > (t'_1, t'_2)$, where t'_1 and t'_2 are sub parse trees with top nonterminal n_1 respectively n_2. The abstract syntax tree is $desugar(< A, c_3 > (t'_1, t'_2)) = c_3(t_1, t_2)$, where $t_1 = desugar(t'_1)$ and $t_2 = desugar(t'_2)$, which is equal to the original abstract syntax tree.

Since every context-free grammar can be rewritten to the Chomsky normal form, the unparser can be defined using a top-down tree-to-string transducer for every context-free grammar. □

The proof that the relation

$$\mathcal{L}(G_{cfg}) \supseteq unparse(desugar(parse(\mathcal{L}(G_{cfg}))))$$

also holds is almost equal to the proof of Theorem 3.4.3. One should take the string s as starting point instead of the abstract syntax tree. The superset relation is a result of the fact that layout is not available in the abstract syntax tree and as a result it cannot be literally restored during unparsing. The language produced by the unparser is thus always a sentence of $\mathcal{L}(G_{cfg})$, but the set of sentences of $\mathcal{L}(G_{cfg})$ is greater than the set of sentences the unparser can produce.

Theorem 3.4.4. The relation $\mathcal{L}(G_{cfg}) \supseteq \mathit{unparse}(\mathit{desugar}(\mathit{parse}(\mathcal{L}(G_{cfg}))))$ holds for the unparser.

(proof) First we show that for each of these three rules in the Chomsky normal form the relation

$$\mathcal{L}(G_{ast}) = \mathit{desugar}(\mathit{parse}(\mathit{unparse}(\mathcal{L}(G_{ast}))))$$

holds when using a context-free grammar without production rules for layout syntax.

Consider the three forms of production rules of the Chomsky normal form. First, consider the production rule $A \rightarrow \epsilon\{c_1\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_1$. The tree-to-string transducer belonging to this production rule is: $q(c_1) \rightarrow \epsilon$. Start with the empty string ϵ. Parsing this string produces the parse tree: $\mathit{parse}(\epsilon) =< A, c_1 > (\epsilon)$. Desugaring this parse tree results in the abstract syntax tree: $\mathit{desugar}(< A, c_1 > (\epsilon)) = t_{ast} = c_1$. Using this abstract syntax tree t_{ast} as input for the unparser results in $q(c_1) = \epsilon$.

Second, consider the production rule $A \rightarrow s\{c_2\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_2$. The tree-to-string transducer belonging to this production rule is: $q(c_2) \rightarrow s$. Parsing the string s produces a parse tree: $\mathit{parse}(s) =< A, c_2 > (s)$. Desugaring this parse tree produces the abstract syntax tree: $\mathit{desugar}(< A, c_2 > (s)) = t_{ast} = c_2$. Using this abstract syntax tree t_{ast} as input for the unparser results in $q(c_2) = s$.

The last production rule is $A \rightarrow \epsilon n_1 \epsilon n_2 \epsilon\{c_3\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_3(t_1, t_2)$, where t_1 and t_2 are the abstract syntax trees belonging to n_1, respectively n_2. The tree-to-string transducer belonging to this production rule is: $q(c_3(x_1, x_2)) \rightarrow \epsilon q_1(x_1) \epsilon q_2(x_2) \epsilon$. Parsing the string $s_1 \cdot s_2$ produces a parse tree $\mathit{parse}(s_1 \cdot s_2) =< A, c > (t'_1, t'_2)$, where t'_1 and t'_2 are sub parse trees with top nonterminal n_1 respectively n_2. The abstract syntax tree is $\mathit{desugar}(< A, c_3 > (t'_1, t'_2)) = t_{ast} = c_3(t_1, t_2)$, where $t_1 = \mathit{desugar}(t'_1)$ and $t_2 = \mathit{desugar}(t'_2)$. Using this abstract syntax tree t_{ast} as input for the unparser results in $q(c_3(t_1, t_2)) = s_1 \cdot s_2$, where $s_1 = q_1(t_1)$ and $s_2 = q_2(t_2)$, i.e. s_1 and s_2 are the result of unparsing the sub-terms t_1 and t_2.

The previous statements show that the relation

$$\mathcal{L}(G_{cfg}) = \mathit{unparse}(\mathit{desugar}(\mathit{parse}(\mathcal{L}(G_{cfg}))))$$

holds when no layout rules are defined in the context-free grammar. Suppose we extend $\mathcal{L}(G_{cfg})$ with layout syntax resulting in $\mathcal{L}(G_{cfg})'$, then $\mathcal{L}(G_{cfg}) \supset \mathcal{L}(G_{cfg})'$, since every sentence without layout must be in $\mathcal{L}(G_{cfg})'$, otherwise the languages are not semantical equal. Thus every sentence the unparser produce must be at least in $\mathcal{L}(G_{cfg})$, otherwise the *unparse* function does not meet the requirement of the unparser to be semantically transparent. □

Theorem 3.4.5. The unparser of Definition 3.3.1 is a linear and deterministic top-down tree-to-string transducer.

(Proof) The derivation of an unparser using Definition 3.3.1 can be mapped on a top-down tree-to-string transducer. Considering Definition 3.3.1 the unparser contains actions of the form:

$$\begin{aligned} unparseA(c) &= s \\ unparseA(c(x_1, \ldots, x_k)) &= s_1 \cdot unparseA'(x_1) \cdot \ldots \cdot s_k \cdot unparseA''(x_k) \cdot s_{k+1} \end{aligned}$$

The similarity of the unparser with the top-down tree-to-string transducer is obvious. Substitute the occurrences of *unparseA* by states named q_A and the unparser is translated to a tree-to-string transducer.

The unparser is linear, since variables are not deleted or copied; each occurrence of x_i is once in the left-hand side and once in the right-hand side.

The unparser is also deterministic, since it is derived from a context-free grammar augmented with signature labels, where the tuples of left-hand side nonterminal n and signature label c are unique. We require that the tuple of (n, c) is unique, so all left-hand sides are unique. □

These theorems show that an unparser can be specified using a linear deterministic top-down tree-to-string transducer. A metalanguage for a template evaluator should be powerful enough to express a tree-to-string transducer to be *unparser*-complete. At that moment the metalanguage can be used to define unparsers for abstract syntax trees belonging to context-free languages, and thus it can be used for instantiating all meaningful sentences of a context-free language.

3.5 Conclusions

We have discussed the relations between concrete syntax, abstract syntax trees and their grammars. The unparser translates an abstract syntax to a concrete syntax and is a meta program instantiating code with two specific properties: parsing and desugaring its output results in the original abstract syntax tree of the used input, and the unparser can instantiate all meaningful sentences of the output language. We showed that a linear deterministic top-down tree-to-string transducer fulfills the requirements to implement an unparser. A metalanguage for this kind of meta programs should at least be powerful enough to express this kind of tree-to-string transducer, otherwise some sentences of the output language cannot be instantiated.

4

The Metalanguage

Our metalanguage for templates is introduced in this chapter. This metalanguage is strong enough to specify unparsers, and still enforces a separation of model and view. We base the constructs of this metalanguage on the theoretical framework of Chapters 2 and 3. The syntax and operational semantics of the constructs are given.

The metalanguage is illustrated by means of an implementation of the PICO unparser. Related template evaluators and their metalanguages are also discussed, including the implementation of an unparser for the PICO language to show the differences between the systems.

4.1 Introduction

In this chapter we present a metalanguage for templates based on the theoretical framework of Chapters 2 and 3. The requirements for our metalanguage are:

- ◇ Strong enough to express the unparser.
- ◇ Minimize the possibilities for expressing calculations in the template.

The first requirement guarantees that the metalanguage forms no limitation on the sentences that templates can instantiate, since an unparser is capable to instantiate all meaningful sentences of its output language.

The second requirement is to enforce separation of model and view. Inspired by [92], our metalanguage should be strong enough to express the view, i.e. unparsers, but it should limit the possibility that model specific code and calculations are specified in the meta program instantiating the code. This is essential to prevent the use of templates for computations which are not part of rendering the view. The availability of a general-purpose metalanguage does not prevent to write complete programs in the template, which breaks the model-view-controller (MVC) architecture. The MVC architecture is discussed in Section 7.2.3.

We start with a formal definition of code generators, a specific class of meta programs instantiating code, in Section 4.2. Section 4.3 discusses our metalanguage and provides a formal specification of its operational semantics. We compare our metalanguage with other template systems in Section 4.5 using a case study based on the implementation of an unparser for the PICO language.

4.2 Code Generators

In Chapters 2 and 3 we already presented two meta programs for instantiating code. The first is the *yield* function, see Definition 2.2.4. The second the *unparse* function, see Definition 3.3.1. The *yield* function reconstructs a sentence from an arbitrary parse tree of an arbitrary context-free language. It is the most generic meta program as it generates a sentence from every arbitrary parse tree. It does not depend on the structure of the tree and only considers the leaves of the tree. The only requirement for the *yield* function in order to get a well-formed output sentence is a well-formed input parse tree.

The *unparse* function is not limited to parse trees and allows abstract syntax trees as input. The abstract syntax tree lacks syntactic information, like layout information, to restore the original code it represents. The *unparse* function contains the missing information in its rules to restore the syntactic sugar missing in the abstract syntax tree. For both the *yield* function and *unparse* function it should be noticed that no semantic information is added to or removed from the generated sentence. Only the representation of the sentence is changed.

The *unparse* function can be considered as a set of small templates, where each unparse equation contains a subtemplate based on the corresponding production rule. For example, consider the right hand side of the unparser for the or case in Example 3.3.3:

$$unparseB(Or(x_1, x_3)) = unparseB(x_1) \cdot \text{“}\,|\,\text{”} \cdot unparseB(x_3).$$

It consists of two recursive calls to the *unparse* function in order to convert subtrees x_1 and x_3 to strings and it contains the lexical representation of the or operator, i.e. |. This meta program transforms an input tree with the *Or* signature to a concrete syntax representation, without altering its meaning.

Just as the *unparse* function, the code generator function *CG* converts a tree into a string and has the signature:

$$CG : Tree \to String$$

The properties of a code generator are defined by the following definition:

Definition 4.2.1. (Code generator). A code generator CG, instantiating sentences of a given $\mathcal{L}(G_{cfg})$ modulo layout, is a function producing at least two sentences of $\mathcal{L}(G_{cfg})$ and at most the set of sentences defined by $\mathcal{L}(G_{cfg}) \setminus \{s\}$, where s is a sentence and $s \in \mathcal{L}(G_{cfg})$, using an input tree language $\mathcal{L}(G_{rtg})$.

Definition 4.2.1 excludes two variants of meta programs instantiating code as a code generator. The first excluded variant are the meta programs producing exactly one sentence of $\mathcal{L}(G_{cfg})$:

$$CG(x) = s, \text{ where } s \in \mathcal{L}(G_{cfg}) \text{ and } x \text{ matches every tree.}$$

No external information is necessary to complete the sentence; it is already complete. All semantic information is *a priori* available in the result sentence of the code generator and the input data has no influence on it. Given that input data is not necessary, the language of the output code is independent of the input data language. When the input data is empty, i.e. no semantic information is specified, this meta program will always generate a sentence s, which means that the level of abstraction of the input data is at the highest level. Since only one sentence s is generated, its functionality domain is limited to the semantics of s.

The second variant of the meta programs excluded by $\mathcal{L}(G_{cfg}) \setminus \{s\}$ in the Definition 4.2.1 is the unparser, as defined in Section 3.3. The unparser has the specific property that the input contains the same semantic information as the output, where a code generator produces a subset of the sentences of $\mathcal{L}(G_{cfg})$. The input language of the unparser is the abstract syntax language belonging to $\mathcal{L}(G_{cfg})$. In that case the code generator is neutral with respect to the behavior and definitions stored in the abstract syntax tree.

Furthermore, for a code generator, the relation

$$\mathcal{L}(G_{rtg}) = desugar(parse(codegenerator(\mathcal{L}(G'_{rtg}))))$$

does not have to hold, and it is not required that G_{rtg} and G'_{rtg} are equal, since the input data can contain less details than the output code. A code generator is not necessarily linear and data may be copied, i.e. it may contain rules of the form $CGA(f(x)) = CGA(x) \cdot CGA(x)$. It must be deterministic, otherwise it can generate different output sentences for a given input tree.

Example 4.2.2. (Code generator). Considering the *unparse* function of Example 3.3.3, once one or more recursive calls in the right-hand side of the equations is substituted by a fixed sentence of terminals or a variable is used more than once in the right-hand side, the unparser becomes a code generator. To illustrate this we applied this to the unparse rule for the or. First by substituting one call by a fixed sentence of terminals:

$$unparseB(Or(x_1, x_3)) = unparseB(x_1) \cdot \text{“ | ”} \cdot unparseB(x_3)$$

is changed to:

$$CGB(f(x)) = \text{``}true\text{''} \cdot \text{``}\,|\,\text{''} \cdot CGB(x)$$

It is no longer possible to use it as *unparse* function, since the set of sentences the code generator can produce is a subset of the sentences belonging to the output language $\mathcal{L}(G_{bool})$. The $unparseB(x_1)$ call can return all sentences belonging to the nonterminal B, while in CGB it is replaced by *"true"*. *"true"* $\subset \mathcal{L}(B)$, so the sentences CGB can produce is a subset of $\mathcal{L}(G_{bool})$.

An example of the second case is the nonlinear code generator:

$$CGB(f(x)) = CGB(x) \cdot \text{``}\,|\,\text{''} \cdot CGB(x)$$

This code generator always produces sentences of the form $s_1|s_2$, where s_1 and s_2 are equal and s_1 and s_2 are sentences of $\mathcal{L}(G_{bool})$, while the unparser allows to generate sentences where s_1 and s_2 are not necessarily equal. $s_1|s_2$, where s_1 and s_2 are equal, are a subset of the sentences where s_1 and s_2 are not necessarily equal, thus this code generator also only produces a subset of sentences of $\mathcal{L}(G_{bool})$.

Further it is not necessary that CGB accepts the abstract syntax language of the output language G_{bool}.

The way an unparser is transformed into a code generator is related to partial evaluation [44]. Informally, some code generators can be obtained by evaluating the unparser using a non-complete abstract syntax tree. The result is a not complete evaluated template.

Considering the semantics, a code generator has the following properties. First, without input data the code generator cannot produce a valid well-formed output sentence, since information is missing. Second, the code generator is not semantic neutral. Contrary to the unparser, the code generator is allowed to add semantic information to or remove semantic information from the input data. As a result the input data tree will only generate a subset of sentences belonging to the output language, or the input data tree can contain more meaningful information than the sentences of the output language can reflect.

Usually a code generator is used to increase the level of abstraction, and fixed blocks of code are mixed with placeholders. As a result of the raise of the level of abstraction, the application domain of the output language is broader than the input data tree language. Informally, a relationship can be found, when pieces of object code in the code generator contain less placeholders, the object code is more complete and contains more semantic information than a code generator closer to an unparser. The input data becomes more abstract with respect to the code it instantiates. If a code generator contains a lot of placeholders, it becomes more like an unparser and the opposite occurs: the code generator adds less semantic information and the input data becomes less abstract with respect to the output sentences.

4.3 Our Metalanguage

The *unparse* function, see Definition 3.3.1, is our starting point for the design of the metalanguage used in our templates. This section discusses our metalanguage constructs, which are based on the requirements for implementing an unparser. Beside the syntax of the constructs, we discuss their formalized operational semantics.

In this chapter we consider a template as a string of characters[1], where it is allowed to have placeholders containing instructions. A placeholder is written between the character sequences `<:` and `:>`; the so called *hedges*. These hedges act as markers to indicate the transition between the object code and the meta code. Since a template is a string with placeholders these hedges are obligatory; otherwise it is not possible to make a distinction between object code and meta code. The syntax of the hedges is free, as long as they are disjoint of character sequences used in the object language.

The evaluation of the placeholders is executed by the template evaluator. It searches for placeholders in the string and replaces them using information from the input data tree. The instructions of the placeholders are evaluated to obtain a string to replace the placeholders. When all placeholders are replaced the evaluator is finished.

Considering the *unparse* function of Section 3.3 we can identify two kinds of instructions. The first type selects an unparse equation based on matching a pattern on a piece of the input data and binds meta-variables to subtrees of it. Second, equations have names of the form *unparseA*, which are called in the right-hand side of the equations having a variable as argument. We offer two constructions which implement this functionality:

◇ Match-Replace (Section 4.3.4);

◇ Subtemplates (Section 4.3.3).

Match-Replace is a mechanism to *match* on input data (sub)trees and depending on a match, returning another (sub) sentence. Subtemplates enable generation of recursive structures, like lists and trees. Further we present derived placeholders, which are abbreviations for constructions using subtemplates and match-replace placeholders:

◇ Substitution (Section 4.3.5);

◇ Conditional (Section 4.3.6);

◇ Iteration (Section 4.3.7).

[1] In Chapter 5 we show that a template is a sentence of a template grammar.

These placeholder constructs are discussed in the coming sections. We discuss the metalanguage by means of a (informal) syntax definition and operational semantics. The operational semantics of the different metalanguage constructs depends on the way templates are evaluated. We start with the discussion of the general function evaluating a template. After the general function is introduced, we describe the syntax and operation semantics of the different placeholder constructs.

4.3.1 Template Evaluation

An important aspect of understanding the metalanguage is the interpretation of the language constructs. For this purpose we introduce the *eval* function, which evaluates a template using some input data resulting in an output sentence. The *eval* function has the following signature:

$$\mathit{eval} : \mathit{Template} \times \mathit{Templates} \times \mathit{MVars} \rightarrow \mathit{String}.$$

The *eval* function has three arguments, the first argument *Template* contains the current template (a string containing placeholders) under evaluation, the other two arguments are context information. The *String* is the result of the template evaluation. For now, we assume that the *eval* function can detect metalanguage code in the template string and call itself recursively to replace these placeholders with strings. We do not discuss this detection of placeholders in detail, since we present an approach based on a combination of object language grammar and metalanguage grammar in Chapter 5.

The context arguments, *Templates* and *MVars*, are symbol tables, where *Templates* contains (sub)templates and *MVars* contains meta-variables. Meta-variables are assigned to (sub)trees of the input data and can be referred in the meta code. The symbol table *Templates* is initialized with all (sub)templates defined at the start of the evaluation. The symbol table *MVars* contains all assigned meta-variables, which is updated during evaluation. Further the *MVars* symbol table is scoped, where each meta-variable is associated with a block, and a meta-variable may only be referred to within the block for which it is associated. Blocks may be nested within other blocks to an arbitrary level, and meta-variables may be referred to anywhere within a block. Meta-variables assigned inside a block override the value of meta-variables with the same name assigned in a parent block. A *scope* of a meta-variable consist of those parts of the program in which a meta-variable may be referred to.

The *eval* function is not directly invoked to start a template invocation, but we use a helper function *start*. This function initializes the context of the *eval* function and has the following signature:

$$\mathit{start} : \mathit{Templates} \times \mathit{InputData} \rightarrow \mathit{String},$$

where *Templates* is a set of (sub)templates stored in a symbol table and *InputData* is a tree representing the input data. Before presenting the equations of the *eval* function, it must be noted that our equations should be interpreted as a conditional term rewriting system [6]. The equations have the form:

$$\frac{\begin{array}{c} t_1 \mapsto t_1' \\ \dots \\ t_i \mapsto t_i' \end{array}}{t \mapsto t'}$$

where $t, t', t_1, t_1', \dots, t_i, t_i'$ are terms. The $\mapsto$ must be read as *results in* or *maps to*. The equation should be then read as "If the rewriting of t_1 results in t_1', ..., of t_i results in t_i' then rewriting of t results in t'". Terms after the $\mapsto$ sign may contain an updated value, used in the remaining (sub) equations. For example, consider the equation:

$$add(bst_{vars}, \text{\$root}, t) \mapsto bst_{vars}$$

If before the equation bst_{vars} equals to $\{\}$, then after evaluation bst_{vars} will hold the value $\{root \mapsto t\}$.

The *start* function is defined as follows:

$$\frac{\begin{array}{c} startblk([\,]) \mapsto bst_{vars1} \\ add(bst_{vars1}, \text{\$root}, t) \mapsto bst_{vars2} \\ add(bst_{vars2}, \text{\$\$}, t) \mapsto bst_{vars3} \\ startblk(bst_{vars3}) \mapsto bst_{vars4} \end{array}}{start(st_{tmps}, t) \mapsto eval(\text{<: template() :>}, st_{tmps}, bst_{vars4})}$$

where st_{tmps} is a symbol table containing the subtemplates, t is the input data treeÂăand bst_{vars} contains the block-structured symbol table for the meta-variables. The functions *startblk*, *add* and *eval*, used in the equation of the *start* function are defined in the following subsections. The *start* function as defined here has some operational consequences: First it assigns the input data tree to the meta-variables $*root* and $$, where $*root* is intended as a global meta-variable containing the original input data tree. The $$ meta-variable is an internal meta-variable which is updated to hold a current context of the input data tree. It has the same behavior as the "normal" meta-variables, but is only implicitly accessible, since the concrete syntax of meta-variables does not allow to write $$ as identifier for a meta-variable. Furthermore the *start* function shows that a template with the name *template* is the starting point of evaluation, since the fixed template in the right-hand side of the *start* function contains a call to that template.

The different equations for the recursive *eval* function are defined by the semantics of the placeholders in the coming subsections. For completeness, the cases for the string without placeholders and the empty string are given by the following equations:

$eval(\epsilon, st_{tmps}, bst_{vars}) \mapsto \epsilon;$

$eval(s, st_{tmps}, bst_{vars}) \mapsto s$, when s does not contain a placeholder.

4.3.2 Block-Structured Symbol Table

The (sub)templates and meta-variables are stored in symbol tables. We use a simple symbol table to store the (sub)templates and a block-structured symbol table for meta-variables. The block-structured symbol table is a simple symbol table extended with the operations for starting and finishing a block. Both kinds of symbol tables are defined in [52]. The operations supported by our symbol tables are:

- ◇ *add* - Adds a meta-variable to a block.
- ◇ *lookup* - Searches for a meta-variable and returns its value.
- ◇ *startblk* - Starts a new block to add meta-variables.
- ◇ *stopblk* - Removes the latest added block of meta-variables.

The standard operations of a symbol table *delete* and *update* are not necessary for our metalanguage, since we do not support reassignment of meta-variables with a new value in a scope and deletion of meta-variables is also not possible.

First we start with discussing the operations and behavior of a simple symbol table. A *symbol table* is modeled by a partial function from symbol, *SYM*, to values, *VAL*:

$$st : SYM \nrightarrow VAL$$

The arrow "$\nrightarrow$" indicates that a function from *SYM* to *VAL* is not necessarily defined for all elements of *SYM* (hence 'partial'). The subset of *SYM* for which the symbol table provides a value is defined as: $dom(st)$. The set of symbols defined by $dom(st)$ is the alphabet Σ_{st} of the symbol table. If a symbol $a \in \Sigma_{st}$, that is $a \in dom(st)$, then $st(a)$ is the unique value associated with a and hence $st(a) \in VAL$. The notation $\{a \mapsto t\}$ describes a function that is only defined for a

$$dom(\{a \mapsto t\}) \mapsto \{a\}$$

which maps a to t

$$\{a \mapsto t\}(a) \mapsto t$$

More generally we can use the notation

$$\{a_1 \mapsto t_1, a_2 \mapsto t_2, \ldots, a_p \mapsto t_p\}$$

where all the a_i's are distinct, to define a function whose domain is

$$\{a_1, a_2, \ldots, a_p\}$$

and whose value for each a_i is the corresponding t_i. For example, if we have a number of meta-variables assigned to a subtree of the input data

$$st = \{\$lhs \mapsto \text{sub(id("repnr"), natcon(1))}, \$natcon \mapsto 1, \$id \mapsto \text{"repnr"}\}$$

The domain of st is $dom(st) = \{\$lhs, \$natcon, \$id\}$ and

$$\begin{aligned} st(\$lhs) &\mapsto \text{sub(id(“repnr"), natcon(1))} \\ st(\$natcon) &\mapsto 1 \\ st(\$id) &\mapsto \text{“repnr"} \end{aligned}$$

The notation $\{\}$ is used to denote the empty symbol table, where $dom(\{\}) = \varnothing$. Initially the symbol table is empty, i.e. $st = \{\}$.

The next step is to introduce block-structured symbol tables. The requirement for a block-structured symbol table is that meta-variables assigned in a parent block can be looked up in a child block and that a meta-variable assigned in a child block can override an earlier assigned meta-variable of its parent block(s). We define a block-structured symbol table as a sequence of simple symbol table, one for each nested block, where functions in the last defined simple symbol table override the functions of earlier instantiated symbol tables. First we give a definition of function overriding.

Definition 4.3.1. (Function overriding) [52]. The operator $\oplus$ combines two functions of the same type to a give a new function.
The new function $f \oplus g$ is defined for an argument a if either

- ◇ $dom(f \oplus g) = dom(f) \cup dom(g)$;
- ◇ $(f \oplus g)(a) = g(a)$, when $a \in dom(g)$;
- ◇ $(f \oplus g)(a) = f(a)$, when $a \notin dom(g)$ and $a \in dom(f)$.

Example 4.3.2. (Function overriding). An example of function overriding is shown by the following equations:

$$\begin{aligned} &\{\$id \mapsto \text{“repnr"}, \$lhs \mapsto \text{sub(id(“repnr"), natcon(1))}\} \\ &\qquad \oplus \{\$lhs \mapsto \text{id(“repnr")}, \$rhs \mapsto \text{natcon(1)}\} \\ &= \{\$id \mapsto \text{“repnr"}, \$lhs \mapsto \text{id(“repnr")}, \$rhs \mapsto \text{natcon(1)}\} \end{aligned}$$

A block-structured symbol table bst is modeled as a sequence of symbol tables st, where the first symbol table st is the outermost block and the last st' is the innermost block. The empty block-structured symbol table is $bst = []$. For example at a given point in the template the bst contains the blocks $bst = [st, st']$, where $st = \{\$id \mapsto \text{“repnr"}, \$lhs \mapsto \text{sub(id(“repnr") natcon(1))}\}$ and $st' = \{\$lhs \mapsto \text{id(“repnr")}, \$rhs \mapsto \text{natcon(1)}\}$. The environment for that point is a single st_{env} obtained by combining all the symbol tables of bst using the equation: $st_{env} = st_1 \oplus \ldots \oplus st_i$, where st_i refers to the innermost block.

The function *env*, with the signature $env : bst \rightarrow st$, obtains an st_{env} from a given *bst* and is defined by the following equations:

$$\begin{aligned} env([]) &\mapsto \{\} \\ env([st]) &\mapsto st \\ env([st_1, st_2, \ldots, st_i]) &\mapsto st_1 \oplus env(st_2, \ldots, st_i) \end{aligned}$$

We will now provide the equations belonging to the functions *add*, *lookup*, *startblk* and *stopblk* operating on a *bst*. The function *startblk*, with the signature $startblk : bst \rightarrow bst$, appends an empty symbol table *st* to *bst* and is defined by the following equations:

$$\begin{aligned} startblk([]) &\mapsto [\{\}] \\ startblk([st_1, \ldots, st_i]) &\mapsto [st_1, \ldots, st_i, \{\}] \end{aligned}$$

The function *stopblk*, with the signature $stopblk : bst \rightarrow bst$, removes the last symbol table *st* to *bst* and is defined by the following equations:

$$\begin{aligned} stopblk([]) &\mapsto [] \\ stopblk([st_1]) &\mapsto [] \\ stopblk([st_1, \ldots, st_{i-1}, st_i]) &\mapsto [st_1, \ldots, st_{i-1}] \end{aligned}$$

The function *add*, with the signature $add : bst \times a \times t \rightarrow bst$, adds a new meta-variable to the last added symbol table and is defined by the following equations:

$$\frac{a \notin dom(st)}{add([st], a, x) \mapsto [st \cup \{a \mapsto x\}]}$$

$$\frac{a \notin dom(st_i)}{add([st_1, \ldots, st_i], a, x) \mapsto [st_1, \ldots, st_i \cup \{a \mapsto x\}]}$$

The function *lookup*, with the signature $lookup : bst \times a \rightarrow t$, uses the given block-structured symbol table to lookup the latest defined value *t* of a given *a* and is defined by the following equation:

$$\frac{\begin{array}{c} env(bst) \mapsto st \\ a \in dom(st) \end{array}}{lookup(bst, a) \mapsto st(a)}$$

The operation *add* has the precondition that the symbol *a* is not present in the innermost symbol table and the operation *lookup* that the symbol *a* is present in the symbol table *env*(*bst*). The rules for handling the case when these preconditions are not met are not presented here. In practice an error must be thrown such as `meta-variable "a" already defined` or `meta-variable "a" not found`. Chapter 8 discusses an approach guaranteeing that these errors are statically detected.

Next sections discuss the syntax and semantics of the placeholder constructs.

4.3.3 Subtemplates

Subtemplates are a mechanism to divide a template in multiple smaller fragments. These subtemplates must each have a unique identifier and can be invoked from other (sub)templates. The first main reason for having subtemplates is to enable recursion; it is possible that a (sub)template can instantiate itself. Recursion is essential to generate tree or list structures. The second reason for a subtemplate mechanism is to reduce the number of code clones in a template definition.

Two constructs are necessary to implement subtemplates. The first is already mentioned by specifying the signature of the *eval* function and *start* function, namely the declaration of subtemplates. The *start* function is invoked with a symbol table of (sub)templates. The concrete syntax of a (sub)template is:

```
 IdCon[ String ],
```

where `IdCon` is the name of the subtemplate and `String` contains the (sub)template and contains output document characters and placeholders. A set of (sub)templates is a list of these declarations, which is mapped to the symbol table st_{tmps} used for the evaluation of the templates. The symbol table st_{tmps} is initialized by mapping subtemplates of the form `IdCon[ String ]` to symbol table functions of the form $a \mapsto t$, where a is equal to the `Identifier` and t is equal to the `String`. The symbol table requires that each a is unique, thus each template must have a unique identifier. The lexical character class for `IdCon` is equal to the character class as defined in Section 2.6.3. The *start* function requires that at least one template is defined with the name `template`.

The second construct is the subtemplate call statement. This placeholder is used in a template and replaced by the result of an evaluated subtemplate. The syntax of a subtemplate call placeholder is:

```
<: IdCon( Expr ) :>,
```

where `IdCon` is the identifier of the called subtemplate and `Expr` contains an expression to set a new value for the context meta-variable $$. The evaluator replaces this placeholder by the result of the evaluated subtemplate with the identifier `IdCon`. Before the subtemplate is evaluated the expression is evaluated to obtain a new context meta-variable $$. The operational semantics of the subtemplate call placeholder is defined by the following equation:

$$\frac{\begin{array}{c} st_{tmps}(idcon) \mapsto s \\ eval(s_1, st_{tmps}, bst_{vars1}) \mapsto s_1' \\ eval(s_3, st_{tmps}, bst_{vars1}) \mapsto s_3' \\ startblk(bst_{vars1}) \mapsto bst_{vars2} \\ add(bst_{vars2}, \$\$, evalexpr(expr, bst_{vars2})) \mapsto bst_{vars3} \\ eval(s, st_{tmps}, bst_{vars3}) \mapsto s_2' \end{array}}{eval(s_1\text{<: } idcon(expr) \text{ :>}s_3, st_{tmps}, bst_{vars1}) \mapsto s_1' \cdot s_2' \cdot s_3'}$$

The expression `Expr` is used to obtain a new value for the internal context meta-variable $$. `Expr` supports the following operations:

- ◇ Meta-variable lookup (`$IdCon`);
- ◇ String constants (`"..."`);
- ◇ Tree path queries (`a1b2`), see Definition 4.3.3;
- ◇ String concatenation (`Expr + Expr`);
- ◇ No operation.

The syntax of the expressions is defined by the following set of context-free production rules:

$$\begin{array}{l} \text{Expr} \rightarrow \text{Expr + Expr} \\ \text{Expr} \rightarrow \text{\$IdCon} \\ \text{Expr} \rightarrow \text{String} \\ \text{Expr} \rightarrow \text{Treequery} \\ \text{Expr} \rightarrow \text{\$IdCon Treequery} \\ \text{Expr} \rightarrow \epsilon \end{array}$$

The string concatenation is only allowed when both expressions reduces to strings, i.e. leaf symbol. The evaluation of the expressions is defined by the following equations:

$$\frac{\begin{array}{c} evalexpr(e_1, bst_{vars}) \mapsto e_1' \\ evalexpr(e_2, bst_{vars}) \mapsto e_2' \\ rank(e_1') = 0 \\ rank(e_2') = 0 \end{array}}{evalexpr(e_1 + e_2, bst_{vars}) \mapsto e_1' \cdot e_2'}$$

$$\frac{c \in \Sigma_{vars} \quad lookup(bst_{vars}, c) \mapsto t}{evalexpr(c, bst_{vars}) \mapsto t}$$

$$\frac{lookup(bst_{vars}, \$\$) \mapsto t \quad evaltreequery(t, tq) \mapsto t'}{evalexpr(tq, bst_{vars}) \mapsto t'}$$

$$\frac{c \in \Sigma_{vars} \quad lookup(bst_{vars}, c) \mapsto t \quad evaltreequery(t, tq) \mapsto t'}{evalexpr(ctq, bst_{vars}) \mapsto t'}$$

$$evalexpr(c, bst_{vars}) \mapsto c$$

$$evalexpr(\epsilon, bst_{vars}) \mapsto lookup(bst_{vars}, \$\$)$$

The tree path queries are (sub) sentences of the path language belonging to the regular tree grammar of the input data. Path languages for regular tree languages are defined by the following definition:

Definition 4.3.3. (Path language) [33]. Let t be a ground term, the path language $\pi(t)$ is defined inductively by:

- ◇ if $t \in \Sigma_0$, then $\pi(t) = t$;
- ◇ if $t = f(t_1, \ldots, t_r)$ then $\pi(t) = \bigcup_{i=1}^{i=r} \{f \cdot i \cdot s | s \in \pi(t_i)\}$.

Example 4.3.4. (Path language) [31]. For $t = a(b(c), a(c, c))$ the path language $\pi(t)$ is $\pi(t) = \{a1b1c, a2a1c, a2a2c\}$.

A tree path query is a (sub) sentence of $\pi(t)$, where t is the tree of the current context meta-variable $$ or the tree obtained from the meta-variable symbol table. The evaluation of a tree path query starts at the root of the tree t and selects a subtree or leaf symbol by sequential stepping down through the tree using the nodes specified in the query. The subtree or leaf symbol where the tree path query points to is returned by the tree path query evaluator for further processing by the expression evaluator.

The input is represented as an ATerm [22], see Section 2.6.3. The ATerm format supports lists, but we do not support selecting an element in these lists; a tree path query may only point to a list node. The evaluation equations for a tree path query are given below:

$$\frac{\begin{array}{c} f = c \\ r \geq i \end{array}}{\mathit{evaltreequery}(f(x_1, \ldots, x_r), \langle c, i, t \rangle) \mapsto \mathit{evaltreequery}(x_i, t)}$$

$$\frac{\begin{array}{c} f = c \\ r \geq i \end{array}}{\mathit{evaltreequery}(f(x_1, \ldots, x_r), \langle c, i \rangle) \mapsto x_i}$$

$$\frac{\begin{array}{c} f = c \\ r < i \end{array}}{\mathit{evaltreequery}(f(x_1, \ldots, x_r), \langle c, i, t \rangle) \mapsto \epsilon}$$

$$\frac{\begin{array}{c} f = c \\ r < i \end{array}}{\mathit{evaltreequery}(f(x_1, \ldots, x_r), \langle c, i \rangle) \mapsto \epsilon}$$

$$\frac{f \neq c}{\mathit{evaltreequery}(f(x_1, \ldots, x_r), \langle c, i, t \rangle) \mapsto \epsilon}$$

$$\frac{f \neq c}{\mathit{evaltreequery}(f(x_1, \ldots, x_r), \langle c, i \rangle) \mapsto \epsilon}$$

$$\mathit{evaltreequery}([x_1, \ldots, x_r], \langle c, i, t \rangle) \mapsto \epsilon$$

$$\mathit{evaltreequery}([x_1, \ldots, x_r], \langle c, i \rangle) \mapsto \epsilon$$

$$\mathit{evaltreequery}([x_1, \ldots, x_r], \langle c \rangle) \mapsto \epsilon$$

$$\frac{c' = c}{\mathit{evaltreequery}(c, \langle c' \rangle) \mapsto c'}$$

$$\frac{c' \neq c}{\mathit{evaltreequery}(c, \langle c' \rangle) \mapsto \epsilon}$$

Note for expressing tree path queries in the *evaltreequery* we mapped a tree path query string to a nested set of tuples of the form $\langle c, i, t \rangle$, $\langle c, i \rangle$ or $\langle c \rangle$, where c is the current node label, i the index and t the tail of the tree path query. For example $a1b1c$ is mapped to $\langle a, 1, \langle b, 2, \langle c \rangle \rangle \rangle$.

```
template[
  Lorem ipsum dolor sit amet, <: sub( ) :>.
  Integer elementum porta facilisis.
]

sub[
  consectetur adipiscing elit
]
```

Figure 4.1 Subtemplate example.

Example 4.3.5. (Expressions). Table 4.1 shows the evaluation result of a number of expressions. We have chosen these examples to demonstrate different kinds of expressions. The tree provided as context for the tree path query evaluation is: $t = a(b(``s_1"), a([c(``s_2"), c(``s_3")]))$.

Example 4.3.6. (Subtemplate placeholder). An example of the declarations of subtemplates is shown in Figure 4.1. The result of this template is after evaluation:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Integer elementum porta facilisis.
```

The `<: sub( ) :>` calls the subtemplate `sub` and inserts the result in the calling template. The call has no expression, since no input data context has to be selected.

4.3.4 Match-Replace

The match-replace placeholder is a construct to specify a finite set of match rules containing a result string. It is a kind of switch-case statement to select

Expression	**Result**
$``a"$	$``a"$
$``a" + ``b"$	$``ab"$
$a1b1$	$``s_1"$
$a1$	$b(``s_1")$
$a2a1$	$[c(``s_2"), c(``s_3")]$
$a2a1c1$	ϵ
$b1$	ϵ
$a1b1 + ``a"$	$``s_1a"$
$a1b1 + a1b1$	$``s_1s_1"$

Table 4.1 Expression examples.

a result string to replace itself. This result string may contain placeholders, which are evaluated before it is used to replace the match-replace placeholder. The syntax of the match-replace is:

```
<: match :>
  <: Matchpattern =:> String
  ...
  <: Matchpattern =:> String
<: end :>
```

As the syntax shows, the match-replace contains a set of match-rules *mr*, which return a string. The match-rules contain a match-pattern and a result string, in equations we use the abstract notation $\langle mp, s \rangle$, where *mp* is the `Matchpattern` and *s* the `String`, for match-rules. We use a simple match-rule selection algorithm, where the first rule with a successful match is selected. It is possible that no match-rule matches the current (sub)tree of the input data. In that case an error is thrown.

Match-rules are matched against the current tree assigned to the internal meta-variable $$. The match-pattern may define meta-variables, which are bound to subtrees of the matched tree and are stored in the current scope of the symbol table. The string of the selected match-rule is evaluated using the symbol table containing the meta-variables assigned during the matching process. The scope of a meta-variable is the string belonging to the successful match-rule, including recursively applied placeholders. Meta-variables in inner blocks hide meta-variables assigned in parent blocks, as the semantics of the block-structured symbol table already defines.

We will now first discuss the evaluation of the match-replace placeholder itself. After that, we formalize the match-patterns and tree matching algorithm. In case of an error the match-replace returns an object *ERROR*. In practice such object can be implemented as an alternative type for string or an exception, and it should hold a message to inform the user.

$$\frac{\begin{array}{c} eval(s_1, st_{tmps}, bst_{vars1}) \mapsto s_1' \\ eval(s_3, st_{tmps}, bst_{vars1}) \mapsto s_3' \\ lookup(bst_{vars1}, \$\$) \mapsto t \\ startblk(bst_{vars1}) \mapsto bst_{vars2} \\ findmatch(t, [mr_1, \ldots, mr_i], bst_{vars2}) \mapsto \langle s, bst_{vars3} \rangle \\ eval(s, st_{tmps}, bst_{vars3}) \mapsto s_2' \end{array}}{eval(s_1 \text{<: match :>} [mr_1, \ldots, mr_i] \text{ <: end :>} s_3, st_{tmps}, bst_{vars1}) \mapsto s_1' \cdot s_2' \cdot s_3'}$$

$$\frac{\begin{array}{c} eval(s_1, st_{tmps}, bst_{vars1}) \mapsto s'_1 \\ eval(s_3, st_{tmps}, bst_{vars1}) \mapsto s'_3 \\ lookup(bst_{vars1}, \$\$) \mapsto t \\ startblk(bst_{vars1}) \mapsto bst_{vars2} \\ findmatch(t, [mr_1, \ldots, mr_i], bst_{vars2}) = \epsilon \end{array}}{eval(s_1\text{<: match :>}[mr_1, \ldots, mr_i] \text{ <: end :>} s_3, st_{tmps}, bst_{vars1}) \mapsto ERROR}$$

$$\frac{match(t, mr_1, bst_{vars}) = \epsilon}{findmatch(t, [mr_1, mr_2 \ldots, mr_i], bst_{vars}) \mapsto findmatch(t, [mr_2 \ldots, mr_i], bst_{vars})}$$

$$\frac{match(t, mr, bst_{vars}) = \epsilon}{findmatch(t, [mr], bst_{vars}) \mapsto \epsilon}$$

$$\frac{match(t, mr_1, bst_{vars}) \mapsto \langle s, bst_{vars} \rangle}{findmatch(t, [mr_1, mr_2 \ldots, mr_i], bst_{vars}) \mapsto \langle s, bst_{vars} \rangle}$$

$$\frac{match(t, mr, bst_{vars}) \mapsto \langle s, bst_{vars} \rangle}{findmatch(t, [mr], bst_{vars}) \mapsto \langle s, bst_{vars} \rangle}$$

The syntax of the match-pattern is similar to the ATerm tree syntax, defined in Section 2.6.3, augmented with syntax for meta-variables. The alphabet of these trees Σ is syntactically limited by `IdCon`, the syntax of the meta-variables is defined as an `IdCon` prefixed with a dollar-sign. The alphabet of meta-variables Σ_{vars} is thus always disjoint from Σ, since the dollar-sign is not allowed for `IdCon`. Note that the internal used meta-variable $\$\$$ is always disjoint of $\Sigma \cup \Sigma_{vars}$, since its syntax is not a sentence of `IdCon` neither of `IdCon` prefixed with a dollar-sign. The rank of a meta-variable is $c \in \Sigma_{vars}$ is $rank(c) = 0$, as it is always a leaf node. It is allowed to have lists in the ATerm tree syntax. These lists are a shorthand notation for binary trees, and are matched via the pattern $[mp_1, \ldots, mp_k]$, where $mp_1, \ldots, mp_k$ are match-patterns. The underlying binary tree structure of lists has as effect that the last match-pattern mp_k in the pattern $[mp_1, \ldots, mp_k]$ is matched against the tail of the list. The tail holds the remaining list or empty list.

The tree pattern matcher is implemented as a root-to-frontier pattern matcher [31]. The matching mechanism is minimalistic and does for example not support associative-commutative matching such as provided by TOM [84]. It tries to match the tree, and during matching it adds the assigned meta-variables to the symbol table bst_{vars}. The *match* function has the signature: $Tree \times Matchpattern \times MVars \rightarrow MVars$. Its operations are defined by the following equations:

$$\frac{\begin{array}{c} f = f' \\ match(x_1, mp_1, bst_{vars1}) \mapsto bst_{vars2} \\ \ldots \\ match(x_r, mp_r, bst_{vars(r)}) \mapsto bst_{vars(r+1)} \end{array}}{match(f(x_1, \ldots, x_r), f'(mp_1, \ldots, mp_r), bst_{vars1}) \mapsto bst_{vars(r+1)}}$$

$$\frac{f \neq f'}{match(f(x_1, \ldots, x_r), f'(mp_1, \ldots, mp_r), bst_{vars}) \mapsto \epsilon}$$

$$\frac{\begin{array}{c} match(x_1, mp_1, bst_{vars1}) \mapsto bst_{vars2} \\ match([x_2, \ldots, x_r], [mp_2, \ldots, mp_k], bst_{vars2}) \mapsto bst_{vars3} \end{array}}{match([x_1, x_2, \ldots, x_r], [mp_1, mp_2, \ldots, mp_k], bst_{vars1}) \mapsto bst_{vars3}}$$

$$\frac{match(x, mp, bst_{vars1}) \mapsto bst_{vars2}}{match([x], [mp], bst_{vars1}) \mapsto bst_{vars2}}$$

$$\frac{match([x_1, \ldots, x_r], mp, bst_{vars1}) \mapsto bst_{vars2}}{match([x_1, \ldots, x_r], [mp], bst_{vars1}) \mapsto bst_{vars2}}$$

$$match([], [], bst_{vars}) \mapsto bst_{vars}$$

$$\frac{\begin{array}{c} c' \notin \Sigma_{mvar} \\ c = c' \end{array}}{match(c, c', bst_{vars}) \mapsto bst_{vars}}$$

$$\frac{\begin{array}{c} c' \notin \Sigma_{mvar} \\ c \neq c' \end{array}}{match(c, c', bst_{vars}) \mapsto \epsilon}$$

$$\frac{\begin{array}{c} c \in \Sigma_{mvar} \\ bst_{vars} \neq \epsilon \end{array}}{match(t, c, bst_{vars}) \mapsto add(bst_{vars}, c, t)}$$

$$\frac{\begin{array}{c} c \in \Sigma_{mvar} \\ bst_{vars} = \epsilon \end{array}}{match(t, c, bst_{vars}) \mapsto \epsilon}$$

The Table 4.2 shows some examples of trees, match patterns and the resulting symbol table. When a match fails the result is ϵ.

Input Data	**MatchPattern**	bst_{vars}
$\text{“}a\text{”}$	$\text{“}b\text{”}$	ϵ
$\text{“}b\text{”}$	$\text{“}b\text{”}$	$\langle\{\}\rangle$
$a(\text{“}b\text{”})$	$a(\$x)$	$\langle\{\$x \mapsto \text{“}b\text{”}\}\rangle$
$a(b(\text{“}c\text{”}), d(\text{“}e\text{”}))$	$a(\$x, d(\$y))$	$\langle\{\$x \mapsto b(\text{“}c\text{”}), \$y \mapsto \text{“}e\text{”}\}\rangle$
$[]$	$[]$	$\langle\{\}\rangle$
$[\text{“}a\text{”}]$	$[\$x, \$y]$	$\langle\{\$x \mapsto \text{“}a\text{”}, \$y \mapsto []\}\rangle$
$[\text{“}a\text{”}, \text{“}b\text{”}, \text{“}c\text{”}]$	$[\$x, \$y]$	$\langle\{\$x \mapsto \text{“}a\text{”}, \$y \mapsto [\text{“}b\text{”}, \text{“}c\text{”}]\}\rangle$
$[\text{“}a\text{”}, \text{“}b\text{”}]$	$[\$x, \$y, []]$	$\langle\{\$x \mapsto \text{“}a\text{”}, \$y \mapsto \text{“}b\text{”}\}\rangle$

Table 4.2 Match pattern examples.

The match-replace placeholder can also be accompanied with an expression:

```
 <: match Expr :>
   <: Matchpattern =:> String
   ...
   <: Matchpattern =:> String
 <: end :>
```

This construction is an abbreviation for a combination of the match-replace placeholder and subtemplates and can be rewritten to the following subtemplate call placeholder:

```
<: id( Expr ) :>
```

and subtemplate:

```
id[
 <: match :>
   <: Matchpattern =:> String
   ...
   <: Matchpattern =:> String
 <: end :>
],
```

where the value of `id` should be unique to prevent collisions with other subtemplates, i.e. the value of `id` should be hygienic [74].

The match-replace is a quite verbose construction for some common operations like substitution, iteration and conditional. The next presented placeholders are abbreviations for the match-replace and are more intuitive for programmers with an imperative background.

4.3.5 Substitution

The substitution placeholder provides a one-to-one insertion of a leaf symbol of the input data tree into the template. The syntax of the substitution placeholder is:

```
<: Expr :>
```

The evaluator replaces the substitution placeholder by a string. The value of this string is obtained by evaluating the `Expr`.

The informal operational semantics of this placeholder are straightforward. The expression is evaluated, which must yield a string, otherwise an error is generated. This result of the expression substitutes the placeholder in the template.

Formally, the evaluation of the substitution placeholder can be written as a combination of subtemplates and match-replace placeholders. Consider Figure 4.2, the substitution placeholder is replaced by a subtemplate call placeholder including its expression. This placeholder sets a new value for the context meta-variable $$. It calls the subtemplate *id*, which iterates over the list of characters in the string, and a string can indeed be mapped to a list of characters. The subtemplate *id′* is called per character, which maps every character in the input data to a character in the object code. The number of match-rules in this mapping is equal to the number of characters supported by the string type of the input data. Note that the names of the subtemplates *id* and *id′* must be unique to ensure they do not conflict with other declared subtemplates.

4.3.6 Conditional

The conditional placeholder selects a result string based on the result of a condition. The syntax of the conditional placeholder is inspired by the if-then(-else) construct that can be found in most imperative languages:

```
<: if Expr == Matchpattern then :>
  String ( <: else :> String )?
<: fi :>
```

The if-then(-else) construct consists of an condition to select the result strings. The condition contains an `Expr` and a `Matchpattern`. The `Expr` is used to calculate a value from the input data. This result of the expression is matched against the `Matchpattern`; when the match is successful the string of the then-part is inserted. In case of an unsuccessful match the else-part is inserted, or when this part is unspecified, nothing is inserted. The chosen string may contain placeholders and is evaluated before inserting it in the template.

```
<: Expr :>
      ⇒
<: id(Expr) :>
id[
 <: match :>
  <: [$mchar, $chars] =:>
   <: id′($mchar) :><: id($chars) :>
  <: [] =:>
 <: end :>]
id′[
 <: match :>
  <: a =:> a
  ...
  <: Z =:> Z
 <: end :>]
```

Figure 4.2 Translation of substitution placeholder to match-replace and subtemplates.

The conditional placeholder can be rewritten to a match-replace placeholder. The translation is defined by the mappings of Figure 4.3 and Figure 4.4. The first mapping is the if-then-else, the second mapping is the if-then. At translation of the if-then, the missing else-part must be defined in the match-replace placeholder. Without this second rule for the empty result, the evaluation of the match-replace placeholder will produce an error when the first match pattern does not match. Since the if-then(-else) is rewritten to a match-replace placeholder, it is possible to have meta-variables in the match pattern of the conditional. During translation the match pattern is only used for the then-part, as a result the meta-variables of that match pattern are only available in that part. The else part uses the default match pattern $\$x$, so $\$x$ is assigned to the result of the expression in case the then part is selected.

4.3.7 Iteration

The iteration placeholder is an abbreviation for the match-replace placeholder for handling lists. It contains an expression to select a list from the input data and it contains a result string, which is instantiated for every element in

<: **if** *Expr* == *Matchpattern* **then** :>
  s_1 <: **else** :> s_2 <: **fi** :>
      $\Rightarrow$
<: **match** *Expr* :>
  <: *Matchpattern* =:> s_1
  <: \$$x$ =:> s_2
<: **end** :>

Figure 4.3 Translation of if-the-else to match-replace.

<: **if** *Expr* == *Matchpattern* **then** :> s <: **fi** :>
    $\Rightarrow$
<: **match** *Expr* :>
  <: *Matchpattern* =:> s
  <: \$$x$ =:>
<: **end** :>

Figure 4.4 Translation of if-then to match-replace.

the list. During iteration the current element is assigned to a user definable meta-variable $*IdCon* in order to use it in the placeholders of the string. The syntax of the iteration placeholder is:

```
<: foreach $IdCon in Expr do :>
  String  ( <: sep :> String )?
<: od :>
```

A separator can be defined in case of a separated list. The mapping of an iteration placeholder to a match-replace placeholder is defined in Figure 4.5 and Figure 4.6. The meta-variable $*IdCon* contains the element of the iteration. The meta-variable $*tail* is bound to the tail of the list and recursively invokes the subtemplate *id* with this new context via a subtemplate call. The identifier of the subtemplate *id* must be unique to remove possible conflicts. The first translation, of Figure 4.5, is for non-separated lists and the second translation, of Figure 4.6, is for separated lists. We need three match-rules to handle the separator in a correct way in the second case. A separator must only be inserted between two elements and is not allowed to terminate a list, which is

$$\begin{array}{l}
\text{<:}\ \textbf{foreach}\ \$\mathit{IdCon}\ \textbf{in}\ \mathit{Expr}\ \textbf{do}\ \text{:>}\ s\ \text{<:}\ \textbf{od}\ \text{:>} \\
\quad\quad \Rightarrow \\
\mathit{id}[\text{<:}\ \textbf{match}\ \mathit{Expr}\ \text{:>} \\
\quad \text{<:}\ []\ \text{=:>}\ \varepsilon \\
\quad \text{<:}\ [\$\mathit{IdCon}, \$\mathit{tail}]\ \ \text{=:>}\ s\ \text{<:}\ \mathit{id}(\$\mathit{tail})\ \text{:>} \\
\text{<:}\ \textbf{end}\ \text{:>}]
\end{array}$$

Figure 4.5 Translation of iteration placeholder to match-replace placeholder (non-separated lists).

$$\begin{array}{l}
\text{<:}\ \textbf{foreach}\ \$\mathit{IdCon}\ \textbf{in}\ \mathit{Expr}\ \textbf{do}\ \text{:>}\ s\ \text{<:}\ \textbf{sep}\ \text{:>}\ s_{sep}\ \text{<:}\ \textbf{od}\ \text{:>} \\
\quad\quad \Rightarrow \\
\mathit{id}[\text{<:}\ \textbf{match}\ \mathit{Expr}\ \text{:>} \\
\quad \text{<:}\ []\ \text{=:>}\ \varepsilon \\
\quad \text{<:}\ [\$\mathit{IdCon}]\ \ \text{=:>}\ s \\
\quad \text{<:}\ [\$\mathit{IdCon}, \$\mathit{tail}]\ \ \text{=:>}\ s\ s_{sep}\ \text{<:}\ \mathit{id}(\$\mathit{tail})\ \text{:>} \\
\text{<:}\ \textbf{end}\ \text{:>}]
\end{array}$$

Figure 4.6 Translation of iteration placeholder with separator to match-replace placeholder (separated lists).

prevented by the second rule. Section 6.8 discusses why this extra rule becomes superfluous in a syntax safe template evaluator.

4.3.8 Unparser Completeness

The presented metalanguage is intentionally minimalistic to prevent it for using for computations other than rendering the view. Although the metalanguage and its used tree pattern matching is minimalistic, the next theorem shows that our metalanguage is unparser-complete:

Theorem 4.3.7. A metalanguage containing constructions for subtemplates and match-replace placeholders is unparser-complete.

(proof) As shown in Lemma 3.4.2, the production rules of each context-free grammar can be mapped on a grammar only containing rules of these forms:

1. $A \to \epsilon\{c\}$, where $A \in N$ and A is the start symbol;
2. $A \to s\{c\}$, where $A \in N$ and $s \in \Sigma^*$;
3. $A \to \epsilon n_1 \epsilon n_2 \epsilon\{c\}$, where $A, n_1, n_2 \in N$.

We show one-by-one that for each of these three forms of production rules the relation

$$\mathcal{L}(G_{ast}) = desugar(parse(unparse(\mathcal{L}(G_{ast}))))$$

holds for an implementation of the *unparse* function using templates.

First, consider the production rule $A \to \epsilon\{c_1\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_1$. The template belonging to this production rule is: $tmp =$

```
template[
<: match :>
 <: c1 =:>
<: end :>
]
```

Evaluating the template using the abstract syntax tree t_{ast} results in

$$start(tmp, t_{ast}) \Rightarrow \epsilon.$$

Parsing this empty string produces the parse tree:

$$parse(\epsilon) \Rightarrow < A, c_1 > (\epsilon).$$

Desugaring this parse tree produces the abstract syntax tree:

$$desugar(< A, c_1 > (\epsilon)) \Rightarrow c_1.$$

Second, consider the production rule $A \to s\{c_2\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_2$. The template belonging to this production rule is: $tmp =$

```
template[
<: match :>
 <: c2 =:> s
<: end :>
]
```

Evaluating the template using the abstract syntax tree t_{ast} results in

$$start(tmp, t_{ast}) \Rightarrow s.$$

Parsing the string s produces a parse tree

$$parse(s) \Rightarrow < A, c_2 > (s).$$

Desugaring this parse tree produces the abstract syntax tree:

$$desugar(< A, c_2 > (s)) \Rightarrow c_2.$$

The last production rule is $A \rightarrow \epsilon n_1 \epsilon n_2 \epsilon \{c_3\}$. The abstract syntax tree belonging to this rule $t_{ast} = c_3(t_1, t_2)$, where t_1 and t_2 are the abstract syntax trees belonging to n_1, respectively n_2. The template belonging to this production rule is: $tmp =$

```
template[
<: match :>
 <: c3( $x1, $x2 ) =:> <: stmp1( $x1 ) :> <: stmp2( $x2 ) :>
<: end :>
]
```

Evaluating the template using the abstract syntax tree t_{ast} results in

$$start(tmp, t_{ast}) \Rightarrow s_1 \cdot s_2,$$

where s_1 is the result of evaluating the subtemplate $stmp_1$ with the input data tree bound to $\$x_1$, and s_2 is the result of evaluating the subtemplate $stmp_2$ with the input data tree bound to $\$x_2$. Parsing the string $s_1 \cdot s_2$ produces a parse tree $parse(s_1 \cdot s_2) \Rightarrow < A, c > (t'_1, t'_2)$, where t'_1 and t'_2 are sub parse trees with top nonterminal n_1 respectively n_2. The abstract syntax tree is $desugar(< A, c_3 > (t'_1, t'_2)) \Rightarrow c_3(t_1, t_2)$, where $t_1 = desugar(t'_1)$ and $t_2 = desugar(t'_2)$, which is equal to the original abstract syntax tree. □

Theorem 4.3.7 shows that not every feature of our metalanguage is necessary to fulfill the requirements to have an unparser-complete metalanguage. First the stack of meta-variables supporting blocks is superfluous when implementing an unparser, since meta-variables are only used in the scope of a match-replace placeholder. However, to use templates for real world code generation scenarios (see case studies of Chapter 7) referring to earlier assigned meta-variables is commonly used, if only just for identifiers based on multiple labels in the input data tree. An example is generating a get function with the name of the class and field: `getNaturalValue`, where `Natural` is the class name and `Value` the field name. The availability of a block-structured stack where earlier assigned meta-variables can be referred limits the size of the input data. It is used to combine information from different nodes stored on different levels in the input data tree. Since the evaluation of a template is top-down, the meta-variables on the stack can only contain information from parent nodes in the input data tree, since the meta-variables assigned in inner blocks are removed when stepping back to outer blocks.

Second for implementing the unparser only expressions supporting meta-variable lookups is necessary, the other operations are for convenience during implementing code generators.

4.4 Example: The PICO Unparser

This section discusses an unparser for the PICO language, see Section 2.6.1, implemented using templates. This unparser implementation shows the application of the subtemplates, match-replace placeholders and substitution placeholders.

The unparser consists of the start template *template* and a subtemplate for each nonterminal used as result symbol of a context-free production rule. The subtemplates are necessary to handle the list structures of the declarations and statements, and the recursive structure of the statements and expressions. They contain a match-replace placeholder to match on the subtree of the abstract syntax belonging to the nonterminal of the sentences the match-replace placeholder instantiates. The number of match-rules of the match-replace placeholders is equal to the number of alternatives belonging to these nonterminals.

This template of Figure 4.7 is an implementation of an PICO unparser and reconstructs the concrete syntax for an abstract syntax tree of PICO. An example of such an abstract syntax tree of a PICO program is shown in Figure 2.3. It is obtained by parsing and desugaring the PICO program of Figure 2.1. This abstract syntax tree can be used as input data for the template to obtain a concrete syntax representation of the PICO program. The layout is not literally restored, since layout information is missing in the abstract syntax tree. The template evaluator uses the layout as it is defined in the template.

4.5 Related Template Systems

Our metalanguage is based on the assumption that a regular tree is used as input data and the resulting text is context-free language. In this section we discuss the metalanguage of some other template systems. We selected three industrially used template evaluators (ERb [58], JSP [12, 100], and Velocity[2]) and an evaluator discussed in the academic literature (StringTemplate [92]). The selection is based on availability of a working template evaluator and to show different metalanguages.

[2] http://velocity.apache.org (accessed on November 30, 2010)

```
template[
<: match :>
<: program( decls($decls), $stms ) =:>
begin declare
<: decls( $decls ) :>;
<: stms( $stms ):>
end
<: end :>
]

decls[
<: match :>
<: [] =:>
<: [ $head ] =:>  <: idtype($head) :>
<: [ $head, $tail] =:> <: idtype($head) :>, <: decls($tail) :>
<: end :>
]

stms[
<: match :>
<: [] =:>
<: [ $head ] =:> <: stm($head) :>
<: [ $head, $tail] =:> <: stm($head) :>; <: stms($tail) :>
<: end :>
]

stm[
<: match :>
 <: assignment( $id, $expr ) =:>
                                  <: $id :> := <: expr( $expr ) :>
 <: while( $expr, $stms ) =:> while <: expr($expr) :> do
                                  <: stms($stms):> od
 <: if( $expr, $thenstms, $elsestms ) =:> if <: expr($expr) :>
       then <: stms($thenstms) :> else <: stms($elsestms) :> fi
<: end :>
]

expr[
<: match :>
 <: natcon( $natcon ) =:> <: $natcon :>
 <: strcon( $strcon ) =:> "<: $strcon :>"
 <: id( $id ) =:> <: $id :>
 <: sub( $lhs, $rhs ) =:> <: expr($lhs) :> - <: expr($rhs) :>
 <: concat( $lhs, $rhs ) =:>
                            <: expr($lhs) :> || <: expr($rhs) :>
 <: add( $lhs, $rhs ) =:> <: expr($lhs) :> + <: expr($rhs) :>
<: end :>
]

idtype[
<: match :>
<: decl( $id, $type ) =:> <: $id :> : <: type($type) :>
<: end :>
]

type[
<: match :>
 <: natural =:> natural
 <: string  =:> string
<: end :>
]
```

Figure 4.7 PICO unparser based on templates.

A metalanguage can be minimalistic, for example only using placeholders containing reference labels. The template evaluator replaces a placeholder with the referred label with a piece of data instead that the placeholder describes an expression. Next to a minimalistic programming language, a language, like Ruby or Java, can be used as metalanguage. Template systems using a rich metalanguage are ERb [58], JSP [12, 100], and Velocity[3]. Furthermore there are systems, like StringTemplate [92], which provide a metalanguage that is somewhere between the label approach and a Turing-complete metalanguage.

We will discuss in the next sections the industrial template evaluators ERb, JSP, and Velocity. They are designed to generate HTML in web applications, although Herrington uses ERb to generate code in a model driven engineering approach [58]. The last discussed template evaluator, StringTemplate, is an academic approach. StringTemplate finds its origin in the application as a template evaluator for a dynamic website[4]. It is used to investigate template evaluators and metalanguage features. For each of these template evaluators we show an implementation of the PICO unparser. In Section 4.5.5 we give a brief evaluation of the differences and similarities between the different metalanguages.

4.5.1 ERb

ERb, discussed in [58], is a text template interpreter for the programming language Ruby[5]. ERb introduces special syntax constructs to embed Ruby code in a text file. The metalanguage of ERb is Ruby and thus Turing-complete, as a result there is no restriction on the code ERb can generate.

The first main construct is <%= `Ruby expression` %>. Its behavior is similar to our substitution placeholder. The Ruby expression is evaluated and the result is emitted to the output. An example of this construct is:
`Hello` <%= `"Jack"` %> which yields `Hello Jack` after evaluation.

The second main construct is <% `Ruby code` %>, which embeds Ruby code in a template. The code is executed, but output text is only emitted in the generated text when print statements are used inside the Ruby code. Ruby statements can span multiple placeholders, so it is possible to use the conditional and iteration statement provided by the Ruby language. This approach to embed Ruby in a template is very flexible. When a new language construct is added to Ruby, it can immediately be used in an ERb template, because ERb does not have any assumptions about the metalanguage, except that it must be Ruby.

An example of an ERb template is presented in Figure 4.8. The first placeholder initializes the array `names`. The second placeholder and the last placeholder

[3] http://velocity.apache.org (accessed on November 30, 2010)

[4] http://www.jguru.com (Accessed on November 30, 2010)

[5] http://www.ruby-lang.org (accessed on November 30, 2010)

```
<%
names = []
names.push({ 'first' => "Jack", 'last' => "Herrington" })
names.push({ 'first' => "Lori", 'last' => "Herrington" })
names.push({ 'first' => "Megan", 'last' => "Herrington" })
%>
<% names.each { |name| %>
Hello <%= name[ 'first' ] %> <%= name[ 'last' ] %>
<% } %>
```

Figure 4.8 ERb example.

are an iteration formed by the Ruby iterator `.each`. The template text between those placeholders is emitted for each element in the array `names`. The same construction of placeholders can also be used for `if` statements and other Ruby language constructs.

Figure 4.9 shows our implementation of the PICO unparser using ERb. The abstract syntax trees of PICO represented as ATerms are mapped one-to-one to an XML representation, such that it can be queried using XPath. The unparser is grouped in three subtemplates: `root`, `statements` and `expr`. The `root` is the starting point of the unparser. At lines 4-9 it generates the variable declarations. Typical for the ERb implementation is the deletion of the quote-sign (") in the query result at line 6, since the XML queries return strings surrounded by quotes. The separator handling using an `if` statement to check for the last element at line 8 is also typical for the ERb implementation.

ERb offers Ruby as metalanguage, but the way Ruby is embedded in ERb results in a limitation of the use of Ruby. Variables in ERb are global accessible and writable in all (sub)templates, which makes out of the box recursive evaluation impossible since earlier assigned variables with the same name in a calling template are overwritten. This is an important difference comparing to our evaluation strategy and metalanguage.

It is obligatory to support recursive template evaluation with scoped variables to have an unparser-complete template evaluator. Since Ruby is offered as metalanguage we solved the problem of global variables in ERb using an explicit stack mechanism in the templates. Consider the `statements` subtemplate, it iterates (line 16) over a list of statements provided by the caller. First it checks if the iterator is in the last cycle at line 17 and pushes the boolean result of the check on the stack. For correct separator handling, this value is popped and used to generate a separator at line 41.

Lines 18–40 contain a case-switch, similar to our match-replace placeholder. It checks the kind of the statement node in the input data to select the corre-

```
root.erb:
begin
  declare
   <% decls = input_data.xpath('//program/decls/list/decl');
               decls.each { |decl| %>
     <%= decl.xpath('value').text.gsub(/\"/, '') %>:
       <%= decl.xpath('*[2]').last.name %>
     <% if decl != decls.last %>,<% end %>
  <% } %>;
  <% last_stm = nil %>
  <% statements = input_data.xpath('//program/list/*') %>
  <%=  subtemplate("statements.erb", binding)%>
end

statements.erb:
<% statements.each { |stm| %>
   <% stack.push stm != statements.last %>
   <% case stm.node_name when "assignment" then %>
         <%= stm.xpath('value').text.gsub(/\"/, '') %> :=
            <% expr = stm.xpath('*[2]');
            print subtemplate("expr.erb", binding) %>
     <% when "while" then %> while
            <% expr = stm.xpath('*[1]');
                print subtemplate("expr.erb", binding) %> do
            <% stack.push statements %>
               <% statements = stm.xpath('*[2]/*'); %>
               <%= subtemplate("statements.erb", binding) %>
               <% statements = stack.pop %> od
     <% when "if" then %> if <% expr = stm.xpath('*[1]');
            print subtemplate("expr.erb", binding) %> then
            <% stack.push statements %>
            <% statements = stm.xpath('*[2]/*'); %>
            <% statements2 = stm.xpath('*[3]/*'); %>
            <% stack.push statements2 %>
            <%= subtemplate("statements.erb", binding) %>
            else
            <% statements = stack.pop %>
            <%= subtemplate("statements.erb", binding) %>
            <% statements = stack.pop %> fi
   <% end %>
   <% if stack.pop then %>;<% end %>
<% } %>

expr.erb:
<% case expr.first.name
    when "natcon" then %> <%= expr.xpath('value').text %>
<%  when "strcon" then %> <%= expr.xpath('value').text %>
<%  when "id" then %> <%= expr.xpath('value').text.gsub(/\"/, '') %>
<%  when "sub" then %> <% exprl = expr.xpath('*[1]');
    stack.push expr.xpath('*[2]'); expr = exprl;
    print subtemplate("expr.erb", binding) %> -
    <% expr = stack.pop; print subtemplate("expr.erb", binding) %>
<%  when "concat" then %> <% exprl = expr.xpath('*[1]');
    stack.push expr.xpath('*[2]'); expr = exprl;
    print subtemplate("expr.erb", binding) %> ||
    <% expr = stack.pop; print subtemplate("expr.erb", binding) %>
<%  when "add" then %> <% exprl = expr.xpath('*[1]');
    stack.push expr.xpath('*[2]'); expr = exprl;
    print subtemplate("expr.erb", binding) %> +
    <% expr = stack.pop; print subtemplate("expr.erb", binding) %>
<% end %>
```

Figure 4.9 PICO unparser implemented using ERb.

sponding concrete syntax. The if statement and while statement recursively contains statements and need to recall the statements subtemplate. When the statements subtemplate is recursively called, the variable statements is pushed on the stack and a new value is assigned to the variable. For example the while at line 22-28, the content of the variable statements is pushed at line 25, a new value is assigned to the variable at line 26 and the subtemplate is called at line 27, finally the original value of the statements variable is reassigned to it at line 28. The subtemplate expr for generating expressions works in a similar way.

4.5.2 Java Server Pages

Java Server Pages (JSP) [12, 100] is a template based system developed by Sun Microsystems. It is designed for generating dynamic web pages and XML messages in Java-based enterprise systems, where the evaluation is tuned for performance. The aim of JSP is to provide separation between the object code and the content generation. The complete Java language is available as metalanguage in JSP pages.

Evaluating a JSP page goes through two phases: a translation phase and a request phase. In a web environment a page is requested via a client. The translation phase is done once per page at the first request. The *JSP compiler* translates the JSP page to a Java servlet class where all object code is embedded in println(...) statements, a kind of a printf-based code generator (Section 1.5.2). This servlet class is instantiated to answer requests to generate the output code, i.e. HTML.

JSP provides two levels of instructions: JSP directives and JSP scripting elements. JSP directives provide information for the translation phase that is independent of any specific request. The scripting elements are the instructions which are executed at every request of the servlet. These scripting elements are comparable to our placeholders.

JSP directives provide global information for instructing the JSP Compiler for creating the servlet class. They have the following syntax:

```
<%@ directive { attr="value" }* %>.
```

Table 4.3 shows the standard directives. Lines 2-5 of Figure 4.10 contain directives to define the basic page settings for the JSP implementation of the PICO unparser. JSP supports tag libraries, Java classes containing functionality, which can be called from a JSP page. In Figure 4.10, we import tag libraries for

Element	Description
`<%@ page ... %>`	Defines page-dependent attributes, such as session tracking, error page, and buffering requirements.
`<%@ include ... %>`	Includes a file during the translation phase.
`<%@ taglib ... %>`	Declares a tag library, containing custom actions, that is used in the page.

Table 4.3 JSP Directives.

Element	Description
`<% ... %>`	Scriptlet, used to embed Java scripting code.
`<%= ... %>`	Expression, used to embed scripting code expressions when the result shall be added to the output code.
`<%! ... %>`	Declaration, used to declare variables and methods in the servlet body of the JSP page implementation class.

Table 4.4 JSP Scripting Elements.

XML querying[6], string manipulation functions[7] and core functions[8].

JSP provides three types of scripting elements, see Table 4.4. The first two are equivalent to the ERb placeholders; one for embedding Java code in a JSP page and one to define expressions, where the output is directly emitted in the result. The third type of placeholder is used to declare variables and methods that get inserted into the main body of the servlet class. It can be used to declare methods, to be called from the template and to specify fields for storing global information independent of single request.

The JSP PICO unparser is shown in Figure 4.10. We discusses the differences with our approach of Section 4.4. First, iteration is supported by a `foreach` statement, as shown at lines 10-16. It generates a list of declarations, separated by a comma. The list separator handling is implemented by a check at line 15, which does not generate a comma at the last cycle of the iteration. The other difference is the variable scopes supported by JSP. In contrast with our metalanguage, JSP does not support a stack where variables are pushed and popped. The requirement for the stack is that variables must be passed to a called (sub)template and variables in the scope of the calling (sub)template should not be overwritten in the called (sub)template. In case of recursion this

[6] http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/x/tld-summary.html (accessed on November 30, 2010)

[7] http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/fn/tld-summary.html (accessed on November 30, 2010)

[8] http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/c/tld-summary.html (accessed on November 30, 2010)

easily happens, when the (sub)template calls itself, the same variable names are used. For example in the `Expr.jsp` subtemplate defined at lines 65-104 of Figure 4.10. JSP offers a scoping mechanism for variables, where we use the *page* scope and *request* scope. The variables in the page scope are only available in the page itself and not in a called (sub)template, while variables in the request scope are also available in the called (sub)templates. An example is shown at lines 81-83. Values extracted from the input data are stored in variables in the page scope, lines 81-82, otherwise they are overwritten by the recursively called (sub)template. Since page scoped variables are not passed to a called (sub)template, they are assigned to a request scoped variable before calling the (sub)template, see lines 83-85.

```
pico.jsp:
<%@ page language="java" contentType="text/plain"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<c:import url="input.xml" var="url" />
<x:parse  xml="${url}" var="input" />
begin
  declare
   <x:forEach var="decl" varStatus="status"
                      select="$input/program/decls/list/decl">
     <x:set var="identifier" select="string($decl/value)"
         scope="page"/>${fn:replace( identifier , "\"", "")}:
         <x:out select="name($decl/*[2])" />
     <c:if test="${status.last=='false'}" >,</c:if>
   </x:forEach>;
   <x:set var="statements" select="$input/program/list"
                                                 scope="request"/>
   <jsp:include page="statements.jsp"/>

end

statements.jsp:
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<x:forEach var="stm" varStatus="status" select="$statements/*">
   <x:choose>
     <x:when select="name($stm)=\"assignment\"" >
      <x:set var="identifier" select="string($stm/value)"
             scope="page"/>${fn:replace( identifier , "\"", "")}
             := <x:set var="expr" select="$stm/*[2]"
             scope="request"/><jsp:include page="expr.jsp"/>
     </x:when>

     <x:when select="name($stm)=\"while\"">
           while <x:set var="expr" select="$stm/*[1]"
           scope="request"/><jsp:include page="expr.jsp"/> do
           <x:set var="statements" select="$stm/*[2]"
           scope="request"/>
           <jsp:include page="statements.jsp"/>
          od
     </x:when>
```

Figure 4.10 PICO abstract syntax tree unparser in JSP.(to be continued)

```
     <x:when select="name($stm)=\"if\"">
     if <x:set var="expr" select="$stm/*[1]" scope="request"/>
                <jsp:include page="expr.jsp"/>then
                <x:set var="statements1" select="$stm/*[2]"
                scope="page"/><x:set var="statements2"
                select="$stm/*[3]" scope="page"/>
                <c:set var="statements" value="${statements1}"
                scope="request" />
                <jsp:include page="statements.jsp"/>
     else
                <c:set var="statements" value="${statements2}"
                scope="request" />
                <jsp:include page="statements.jsp"/>
     fi
     </x:when>
   </x:choose>
   <c:if test="${status.last=='false'}" >;</c:if>
</x:forEach>

expr.jsp:
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<x:choose>
 <x:when select="name($expr)=\"natcon\"" >
  <x:out select="$expr/*[1]"/>
 </x:when>
 <x:when select="name($expr)=\"strcon\"" >
  <x:out select="$expr/*[1]"/>
 </x:when>
 <x:when select="name($expr)=\"id\"" >
  <x:set var="identifier" select="string($expr/*[1])"
  scope="page"/>
  ${fn:replace( identifier , "\"", "")}
 </x:when>
 <x:when select="name($expr)=\"sub\"" >
  <x:set var="exprl" select="$expr/*[1]" scope="page"/>
  <x:set var="exprr" select="$expr/*[2]" scope="page"/>
  <c:set var="expr" value="${exprl}" scope="request" />
  <jsp:include page="expr.jsp"/> - <c:set var="expr"
  value="${exprr}" scope="request" />
  <jsp:include page="expr.jsp"/>
 </x:when>
 <x:when select="name($expr)=\"concat\"" >
  <x:set var="exprl" select="$expr/*[1]" scope="page"/>
  <x:set var="exprr" select="$expr/*[2]" scope="page"/>
  <c:set var="expr" value="${exprl}" scope="request" />
  <jsp:include page="expr.jsp"/> || <c:set var="expr"
  value="${exprr}" scope="request" />
  <jsp:include page="expr.jsp"/>
 </x:when>
 <x:when select="name($expr)=\"add\"" >
  <x:set var="exprl" select="$expr/*[1]" scope="page"/>
  <x:set var="exprr" select="$expr/*[2]" scope="page"/>
  <c:set var="expr" value="${exprl}" scope="request" />
  <jsp:include page="expr.jsp"/> + <c:set var="expr"
  value="${exprr}" scope="request" />
  <jsp:include page="expr.jsp"/>
 </x:when>
</x:choose>
```

Figure 4.11 PICO abstract syntax tree unparser in JSP.(continued)

Statement	Description
`#set`	Assigns a value to a variable.

`#if( Expression )...`
`#elseif( Expression )`
`...#else...#end`
	Selects a text based on the result of the expression.

`#foreach( Expression )...#end`	Iterates over a list returned by the expression and instantiates the text for each element in the list.
`#literal()...#end`	Disables the evaluator for a section.
`#include( filename )`	Includes the file named `filename` without interpretation of meta code.
`#parse( filename)`	Includes the file named `filename` with interpretation of meta code.
`#stop`	Stops of the evaluator.
`#evaluate`	Evaluates a string containing meta code.
`#define`	Assigns a block of VTL to a reference.
`#macro`	Defines a repeated segment of a VTL template.

Table 4.5 VTL Statements.

4.5.3 Velocity

Velocity[9] is a template evaluator for Java. It provides a basic template metalanguage to reference Java objects, called Velocity Template Language. Velocity is text-based and has no assumptions on the language of the code it generates. The aim of Velocity is to separate the presentation tier and the business tier, according to the model-view-controller (MVC) architecture in web applications. The metalanguage of Velocity supports complex computations, since it is possible to assign variables and write expressions.

A few metalanguage instructions and an example of a Velocity template are already discussed in Section 1.5.4. The Velocity Template Language (VTL) has two core notations: object references written as a $ followed by its variable name, and statements starting with a # followed by the instruction. The object references can be directly used similar to the substitution placeholder or as variables in the statements. The VTL statements are shown in Table 4.5.

The Velocity PICO unparser is shown in Figure 4.12. Velocity provides internal handling of XML data and the metalanguage is able to express the unparser in a compact definition. The difference between the Velocity PICO unparser and the implementation based using our metalanguage is the use of the `foreach` statement and `if`. However, Velocity has the same problem with the variable scopes as JSP. As a consequence temporary variables are necessary when a subtemplate is recursively called multiple times, for example at lines 48-49.

[9] http://velocity.apache.org (accessed on November 30, 2010)

```
begin
 declare
  #foreach ($decl in
   $root.getRootElement().getChild("decls")
                                .getChild("list").getChildren() )
   $decl.getChild("value").getText().replace('"', ''):
   $decl.getChildren().get(1).getName()
   #if( $velocityHasNext ),#end
   #end;
   #set( $statements = $root.getRootElement().getChild("list") )
   #statements(  $statements )
end

#macro (statements $statements)
#foreach ( $stm in $statements.getChildren() )
#if( $stm.getName() == "assignment" )
    $stm.getChild("value").getText().replace('"', '') :=
    #set($expr = $stm.getChildren().get(1) )#expr($expr)
#elseif( $stm.getName() == "while" )
    while #set($expr = $stm.getChildren().get(0) )
          #expr($expr) do #set($statements =
          $stm.getChildren().get(1))#statements( $statements )
    od
#elseif( $stm.getName() == "if" )
  #set($stms1 = $stm.getChildren().get(1) )
     #set($stms2 = $stm.getChildren().get(2) )
    if #set($expr = $stm.getChildren().get(0) )
        #expr($expr) then
        #set($statements = $stms1)#statements(  $statements )
    else
        #set($statements = $stms2)#statements(  $statements )
    fi
#end
#if( $velocityHasNext );#end

#end
#end


#macro (expr $expr )
#if( $expr.getName() == "natcon" )
  $expr.getValue()
#elseif( $expr.getName() == "strcon" )
  $expr.getValue()
#elseif( $expr.getName() == "id" )
  $expr.getValue().replace('"', '')
#elseif( $expr.getName() == "sub" )
  #set($exprl = $expr.getChildren().get(0) )
  #set($exprr = $expr.getChildren().get(1) )
  #expr($exprl)-#expr($exprr)
#elseif( $expr.getName() == "concat" )
  #set($exprl = $expr.getChildren().get(0) )
  #set($exprr = $expr.getChildren().get(1) )
  #expr($exprl)||#expr($exprr)
#elseif( $expr.getName() == "add" )
  #set($exprl = $expr.getChildren().get(0) )
  #set($exprr = $expr.getChildren().get(1) )
  #expr($exprl)+#expr($exprr)
#end
#end
```

Figure 4.12 PICO abstract syntax tree unparser in Velocity.

Statement	Description
`$attribute$`	Attribute references. For example: `$user.name$`.

`$if(attribute)$subtemplate`
`$else$subtemplate2$endif$`
	Conditional template inclusion. For example: `$if(attr)$<title>$attr$</title>$endif$`.

`$template(argument-list)$`	(Recursive) template references. For example: `$item()$`.
`attribute:{anonymous-template}`	Template application to a multi-valued attributes, i.e. iteration. For example: `$users:{<tr><td>$it.name$</td><td> $it.age$</td></tr>}`.

Table 4.6 StringTemplate statements.

4.5.4 StringTemplate

StringTemplate [92] is a text template system designed to enforce strict separation of model and view in a model-view-controller architecture. The developers of StringTemplate have designed it via an evolution process starting from a simple "document with holes" to a sophisticated template evaluator. This separation of model and view is achieved by enforcing that the view cannot modify the model or perform calculations based on data from the model. Therefore, in comparison to JSP or ERb, the design of StringTemplate is optimized for enforcement of separation, not for Turing completeness, nor amazingly-expressive "one-liners" [92].

This limited metalanguage enforces a template developer to exclude any calculations in the template and enforces to only consider the output code. It supports attribute references, subtemplates, implicit `for`-loops and `if` constructs. The authors of StringTemplate distilled four important metalanguage constructs, shown in Table 4.6.

This set of constructs is, in their experience, powerful enough to specify templates for complex dynamic websites and they use StringTemplate for www.jguru.com[10]. An interesting similarity with grammars is discovered. The analogy with grammars is that (sub)templates are production rules and the attribute references are the terminals. The result of this observation is that the explicit `for`-loop construction is not necessary to generate lists. Instead of a `for`-loop, a kind of regular expression notation can be used, like `$names:item()$`, where the subtemplate `item()` is invoked for every element of the list `names`. Although the notation is shorter, the behavior is equal to an iteration placeholder.

Figure 4.13 shows the PICO unparser specified in StringTemplate. The compact notation for the iterator results in a short definition of the unparser when compared to the lines of code of the other unparser specifications. The handling of the input data in StringTemplate is not as flexible as the other systems.

[10] accessed on November 30, 2010

StringTemplate uses a Java Map structure as input data, which we instantiate via a JavaScript Object Notation (JSON)[11][35] of our abstract syntax tree. JSON trees differ from the trees used in this thesis. JSON nodes are a representation of JavaScript objects with fields, such fields do not have an index and are only accessible via their labels. Fortunately, JSON supports ordered lists and as a workaround we emulate the ordered trees by pushing the children of a node into a list. Consider an ATerm of the form $f(t_1, \ldots, t_r)$, which is translated to a JSON node of the form $\{``f" : [t_1, \ldots, t_r]\}$, where the branches are stored in an ordered list, which can be queried using the location of an element. When t_i is a list, the translation will result in a nesting of listings, which cannot be queried by the mechanism of StringTemplate.s Therefore, we translate an ATerm t_i to $\{``list" : t_i\}$, when $t_i$ is a list. Alternatively we could use index labels for the branches, for example $f(t_1, \ldots, t_r)$ to $\{``f" : \{``one" : t_1\}, \ldots, \{``r" : t_r\}\}$. The last translation is mapping the keyword `if` in the abstract syntax tree to `when`, since `if` is a keyword in StringTemplate and thus not allowed as label reference.

Furthermore, StringTemplate does not allow to obtain an element of a list based on its index, but provides standard list functions `first`, `last` and `rest`. `first` returns the first element, `last` the last element of the list and `rest` the original list without the first element. We use the `first` and `last` functions to directly request an element of the list when possible, for the other arguments we obtain them using the `rest` function. If we need the element at index k, where $k > 1$, we apply the `rest` function $k - 1$ times and take the first element of the last `rest` call: `first(rest( ... rest( list ) ... ))`. Line 23 of Figure 4.13 shows this approach.

4.5.5 Evaluation

We presented the PICO unparser implemented in ERb, JSP, Velocity and StringTemplate. We discussed the notable differences and similarities.

StringTemplate and our metalanguage do not allow manipulation of data, and as a result calculations cannot be expressed. The metalanguages of the industrial approaches, ERb, JSP and Velocity, are Turing complete. A Turing complete metalanguage allows manipulating of data and storing of data. This is unnecessary to implement an unparser and can only lead to undesired programming in templates. Programming in templates can result in undesired tangling of concerns [62].

Although the metalanguages of the three industrial approaches are Turing complete, these systems share the same problem. They support recursion, but the assigned meta-variables are by default globally available instead of scoped for the block where they are defined. We used some workarounds to enable

[11] http://www.json.org

```
picounparser.st:
begin
  declare
  $first(program).decls:decl(); separator=","$;
  $last(program).list:statements(); separator=";"$
end

decl.st:
$first(it.decl).value$:$last(it.decl).keys$

statements.st:
$if(it.assignment)$

$first(it.assignment).value$ :=
  $expr(expr=last(it.assignment))$
$elseif(it.while)$

while $expr(expr=first(it.while))$ do
  $last(it.while).list:statements(); separator=";"$
od
$else$
if $expr(expr=first(it.when))$ then
  $first(rest(it.when)).list:statements(); separator=";"$
else
  $last(it.when).list:statements(); separator=";"$
fi
$endif$

expr.st:
$if(expr.natcon)$
$expr.natcon$
$elseif(expr.strcon)$
"$expr.strcon$"
$elseif(expr.id)$
$expr.id$
$elseif(expr.sub)$
$expr(expr=first(expr.sub))$-$expr(expr=last(expr.sub))$
$elseif(expr.concat)$
$expr(expr=first(expr.concat))$||$expr(expr=last(expr.concat))$
$elseif(expr.add)$
$expr(expr=first(expr.add))$+$expr(expr=last(expr.add))$
$endif$
```

Figure 4.13 PICO abstract syntax tree unparser in StringTemplate.

block scoping of meta-variables, but it introduced some undesired boilerplate code. In case of JSP and ERb an explicit stack mechanism for the meta-variables must be defined. In case of Velocity we used helper meta-variables to prevent updates of meta-variables from an inner scope.

The PICO unparser based on StringTemplate has the fewest lines of code. StringTemplate provides a block scoping mechanism making explicit definition of a stack unnecessary. However, in comparison to our metalanguage, StringTemplate has the limitation that it can only handle unordered trees. An extra transformation is necessary to convert the input data from an ordered tree to an unordered tree. This conversion uses ordered lists to represent the unordered children of a node, where the StringTemplate meta code can fetch an indexed element using a number of list operations.

Besides the metalanguage, the (implicit) behavior of the template evaluator is important. During implementation of the unparser, we experienced that, for example, some versions of ERb print text to the output at spurious moments. This is not a problem when having a single template, but with the recursively called subtemplates, the output characters are printed in the wrong order. Subtemplates must be first evaluated and the resulting string must be inserted in the calling template and not directly be sent to the output stream.

4.6 Conclusions

In this chapter we defined our metalanguage based on the linear deterministic top-down tree-to-string transducer. Five constructs are provided: subtemplates, match-replace placeholder, substitution placeholder, iteration placeholder and conditional placeholder. The support for subtemplates and match-replace placeholders ensures that our metalanguage is unparser-complete. The substitution placeholder, iteration placeholder and conditional placeholder are abbreviations for combinations of subtemplates and match-replace placeholders. Our metalanguage cannot change the input data to prevent calculations in the template and enforce separation of model and view.

We implemented unparsers for the PICO language to compare our metalanguage with the metalanguages of ERb, Velocity, JSP and StringTemplate. StringTemplate needs the fewest lines of code to implement an PICO unparser, but in contrast with our metalanguage, StringTemplate cannot directly accept all regular trees without a translation function. The ERb, Velocity, JSP come with a Turing complete metalanguage. These rich metalanguages increase the chance of undesired programming in templates, which can result in tangling of concerns. Although these industrial approaches provide a Turing complete metalanguage they do not have a block scoping mechanism for the meta-variables. A workaround for proper handling of meta-variable scopes was necessary to implement the PICO unparser. This resulted in additional boilerplate code.

5 Syntax Safe Templates

Text–based template evaluators only contain a parser for the metalanguage, but the object language is considered as text. We discuss the construction of a template grammar aware of (sub) languages in a template. Parsing a template using a template grammar ensures that all (sub) sentences are syntactically correct. The construction of such a template grammar is generic and based on the combination of the metalanguage grammar and the object language grammar, where only a combination grammar connecting both has to be defined manually. Presence of a template grammar allows one to detect misspellings in the object language and metalanguage during parsing, without the need of compiling or interpreting generated code.

5.1 Introduction

Writing templates, and code generators in general, is a complex and error prone task. This complexity[1] mainly results from mixing multiple languages in a template, executed at different stages, and the incompleteness of the object code. Manual verification of incomplete object code is hard to do and computers cannot execute incomplete code. Hence, generating all possible outputs followed by verifying the result using a compiler or interpreter seems a valid route. However, to guarantee that a template generates syntactically correct sentences during production use, a possibly exploding amount of input data test cases must be defined for every template. Furthermore, an error must be manually traced back to its origin, which is not always obvious, such as sometimes experienced when using the C preprocessor [41], where the compiler error messages point to the post-processed code instead of the original source

[1] The complexity here considered is complexity in the broadest sense of the word and not for example computational complexity.

code. Checking the template directly offers accurate error messages, pointing to the origin of the error.

Text-based template evaluators are not able to check the object code, since they have no notion of the object language. These evaluators only process and check the meta code and do not deal with the correctness of the rest of the template. Ignorance of the correctness of the object code can lead to undetected syntax errors [103]. Misspellings in the object code, such as missing semicolons, are easily made and in such case text-based template evaluators generate syntactically incorrect code without giving a warning.

Beside the problems during development, dynamic text-based code generation as used in web applications can result in serious security issues, like malicious code injection. We will discuss an example of malicious code injection in Chapter 7.

In order to remove the possibility of syntax errors in the generated code, we introduce the notion of *syntax safety* for templates. Syntax safety is a property of a code generator. For every possible input the output of a syntax safe code generator can be recognized by a parser for the intended resulting language, i.e. the code generator produces output sentences of the language $\mathcal{L}(G_{intended})$. The intended language is the language for which the code generator should produce sentences, for example, Java or C.

We present an approach based on constructing a grammar for templates containing the definition of the metalanguage and the object language of a template. The benefit of our approach is that not only the output code is syntactically correct, but also syntax errors are found in the template itself. Hence, this approach helps to avoid syntax errors, both in the meta code and in the object code, before the template is used for generating code.

Figure 5.1 shows the architecture used for syntax safe template evaluation. The first stage is parsing the template using a template grammar. This chapter focuses on specifying template grammars based on an (off-the-shelf) grammar of the object language [70]. The second stage is static semantic checking, which is discussed in Chapter 8. The static semantic checking is optional as it is not necessary to achieve syntax safety. The last step is the evaluation of templates. We implemented a syntax safe template evaluator called *Repleo*. Repleo is a generic syntax safe template evaluator system parameterized with the template grammar. Chapter 6 discusses this evaluator.

5.2 Syntax Safe Templates

During the discussion of the metalanguage for templates in Chapter 4, we ignored the object language of templates. The object language was considered as strings or sequences of alphabet symbols. Considering the object language as strings does not guarantee that the output sentences are well-formed with

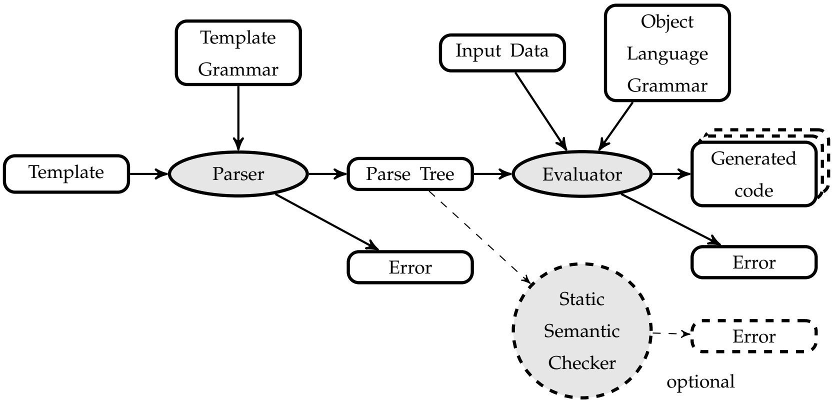


Figure 5.1 Syntax safe evaluation architecture.

respect to their intended output language $\mathcal{L}(G_{intended})$. This section will discuss the requirements to ensure that a template cannot produce sentences that are not in $\mathcal{L}(G_{intended})$.

In a template the following languages are involved: $\mathcal{L}(G_{object})$, $\mathcal{L}(G_{meta})$, $\mathcal{L}(G_{template})$ and $\mathcal{L}(G_{output})$. The sentences and structure of the template directly copied to the output sentence of the evaluator, i.e. the object code, are defined by the object language $\mathcal{L}(G_{object})$. The placeholders in the template are sentences of the (context-free) metalanguage $\mathcal{L}(G_{meta})$ and they are interpreted by the template evaluator and not propagated to the output sentence. The template is a sentence of $\mathcal{L}(G_{template})$, which is based on the union of the grammars of G_{object}, G_{meta} and production rules connecting both. The sentences produced by a template form the output language $\mathcal{L}(G_{output})$.

For example, given the template `<: $x :>b<: $y :>`, where `$x` and `$y` are meta-variables bound to `$x = a` respectively `$y = c`. Evaluating this template results in the sentence `abc`. Considering the different languages, the following statements must hold for the grammars describing this template:

- ◇ `abc` is a sentence of $\mathcal{L}(G_{output})$;
- ◇ `<: $x :>` and `<: $y :>` are sentences of $\mathcal{L}(G_{meta})$;
- ◇ `b` and `n1n2n3`, where `n1`, `n2` and `n3` are nonterminals, are sentences of $\mathcal{L}(G_{object})$. `n1n2n3` represents the structure defined by the object language grammar. The concrete implementation of grammar G_{object} is undefined as long as it produces a language with a sentence of the form `n1n2n3`. Furthermore, since the template contains a `b` on the place of `n2` the language $\mathcal{L}(n_2)$ must at least contain the sentence `b`;

- ◇ `<: $x :>b<: $y :>` is a sentence of $\mathcal{L}(G_{template})$.

We have discussed $\mathcal{L}(G_{object})$, $\mathcal{L}(G_{meta})$, $\mathcal{L}(G_{template})$ and $\mathcal{L}(G_{output})$ from a descriptive point of view. Since we aim for syntax safe templates, we will discuss these languages from a declarative point of view, where the languages describe the allowed sentences. First a new language $\mathcal{L}(G_{intended})$ is introduced. This language is the intended output language of a code generator, where we assume that $\mathcal{L}(G_{intended})$ is a context-free (programming) language. It is undesirable that a meta program can produce sentences which are not part of $\mathcal{L}(G_{intended})$; a meta program should be syntax safe.

Definition 5.2.1. (Syntax safety). A meta program p is called syntax safe with respect to the intended language $\mathcal{L}(G_{intended})$ if p, independent of its input, always generates a sentence s which is a sentence of the intended language $\mathcal{L}(G_{intended})$. In other words program p is syntax safe if $\mathcal{L}(G_{output}) \subseteq \mathcal{L}(G_{intended})$.

We aim for syntax safe template evaluation, therefore we want to construct a $G_{template}$ describing the templates which always result in generation of a sentence in $\mathcal{L}(G_{intended})$. A template is a sentence of $\mathcal{L}(G_{template})$, which is based on a combination of the grammars G_{object} and G_{meta}. The next lemma shows that $\mathcal{L}(G_{object})$ is at least equal to $\mathcal{L}(G_{output})$.

Lemma 5.2.2. The output language of a template $\mathcal{L}(G_{output})$ is a superset of the object language $\mathcal{L}(G_{object})$.

(proof) Given a template *template*$[s]$ without placeholders, then
$s \in \mathcal{L}(G_{object})$. The object code is copied to the output of the template evaluator $start(s,\epsilon) \Rightarrow s$, so s must be a sentence of the language $\mathcal{L}(G_{output})$. So all sentences of $\mathcal{L}(G_{object})$ must be in $\mathcal{L}(G_{output})$. Beside the templates without placeholders, the template may contain placeholders. Placeholders can be substituted by any sentence, which does not have to be specified by the object language, as a result $\mathcal{L}(G_{output}) \supseteq \mathcal{L}(G_{object})$. □

In case of a text-based template environment $\mathcal{L}(G_{object})$ is defined as all possible sentences using an alphabet (most times the set of ASCII characters) minus the syntax defined by $\mathcal{L}(G_{meta})$. Considering the example `<: $x :>b<: $y :>`, the placeholders `<: $x :>` and `<: $y :>` are not defined by the object language, otherwise the template evaluator cannot make a distinction between meta code and object code. The $\mathcal{L}(G_{object})$ used in text-based templates has no restrictions, since it is only defined as a sequence of alphabet symbols. The result is that the output language $\mathcal{L}(G_{output})$ of text-based templates is often a superset of the intended output language $\mathcal{L}(G_{intended})$. This $\mathcal{L}(G_{object})$ enables the template evaluator to generate a sentence that is not in the set of sentences produced by $\mathcal{L}(G_{intended})$ and thus can result in generating code containing syntax errors.

For example in case of generating code for the PICO language, it is possible in a text-based template environment to generate an identifier starting with a number, which is not allowed by the PICO language.

In order to satisfy the requirements for syntax safety, it is necessary to ensure that the object code in a template exist of (sub)sentences of $\mathcal{L}(G_{intended})$ and that the meta code is replaced by (sub)sentences of $\mathcal{L}(G_{intended})$. The approach to guarantee that placeholders are substituted by (sub) sentences of $\mathcal{L}(G_{intended})$ is discussed in Chapter 6.

In order to ensure the object code of a template is a (sub) sentence of $G_{intended}$, a template grammar $G_{template}$ is constructed defining all valid templates for $\mathcal{L}(G_{intended})$. The G_{object} part of $G_{template}$ should be equal to $G_{intended}$ and the start symbol of $G_{template}$ should also be equal to the start symbol of $G_{intended}$. Indeed, $\mathcal{L}(G_{template})$ should also contain the sentences without placeholders, i.e. sentences of $\mathcal{L}(G_{intended})$.

Merging both grammars is not sufficient, as a connection between both grammars must be made. Otherwise the sentences of the metalanguage are not allowed as sub sentences of the object language. A context-free language can be extended by adding new alternatives for nonterminals. When a template contains placeholders, G_{object} should be extended with the placeholder syntax of G_{meta} resulting in a $G_{template}$ in such way that the template is a sentence of that instantiated $G_{template}$.

The placeholder syntax defined in G_{meta} is added as an alternative for nonterminals of G_{object} in $G_{template}$. Hence, the placeholders in syntax safe templates can only replace context-free parts, as it cannot cover multiple nonterminals. For example, in a text-based template it is possible to write the PICO template `BEG<: id() :>ND`, while in a template based on a context-free grammar it is only possible to write `BEGIN <: id() :> a := 1 END`. In the first case the placeholder overlaps multiple (non)terminal symbols, while in the second case the placeholder can be parsed as the nonterminal *DECLS*. The next theorem shows that a $G_{template}$ can be constructed ensuring that the object code and its output language is a (sub)sentence of $\mathcal{L}(G_{intended})$.

Theorem 5.2.3. For every $\mathcal{L}(G_{intended})$ a template grammar $G_{template}$ containing placeholder syntax can be constructed, which ensures that the object code is a (sub)sentence of that $\mathcal{L}(G_{intended})$.

(proof) Given a template $tmp = template[s]$, evaluating this template results in the output sentence $start(tmp, \epsilon) \Rightarrow s$. The template is syntax safe if $s \in \mathcal{L}(G_{intended})$, i.e. parsing s using $G_{intended}$ should return a parse tree. So $G_{template}$ must be a sub grammar or equal to $G_{intended}$, otherwise it is possible to generate a sentence s not in $\mathcal{L}(G_{intended})$.

A template can contain placeholders. They are not part of the syntax of the object language. In order to ensure syntax safety, under a strict condition

$G_{template}$ may be extended with the syntax of the placeholders. This condition is that all additional placeholder syntax is replaced by valid (sub) sentences of $G_{intended}$ during evaluation of the template. We have two kernel placeholders: the subtemplate call placeholder and the match-replace placeholder. For both placeholder we show that extending nonterminals in $G_{template}$ is syntax safe.

Assume that $G_{template}$ contains an object language production rule of the form $S \rightarrow AA'$, where S is the start symbol and A and A' are nonterminals. Consider the templates *tmps*:

```
template[
 <: a() :> s'
]
a[
 s''
]
```

where `s'` and `s''` may recursively contain placeholders. The evaluation

$$start(tmps, t) \Rightarrow s$$

is syntax safe, when for every input data (sub)tree t accepted by the template the output $s \in \mathcal{L}(G_{intended})$. That is, when `<: a() :>` replaces itself by sentence of $\mathcal{L}(A)$ and $s' \in \mathcal{L}(A')$. Subtemplate `a` must produce a sentence of $\mathcal{L}(A)$ to ensure that `<: a() :>` replaces itself by sentence of $\mathcal{L}(A)$, which means that the production rule $TMPS \rightarrow “a[”A“]”$ is in $G_{template}$, where $TMPS$ is the nonterminal representing the list of (sub)templates. Since the subtemplate `a` produces a sentence of $\mathcal{L}(A)$ it is allowed to add the production rule

$$A \rightarrow “\langle{:}\ a(”Expr“)\ {:}\rangle”$$

in $G_{template}$, where $Expr$ is the nonterminal for the metalanguage expressions.

Assume that $G_{template}$ contains object language production rules $S \rightarrow A$ and $S \rightarrow A'$, where S is the start symbol and A and A' are nonterminals. Consider the template *tmps*:

```
template[
 <: match :>
   <: mp =:> s'
   <: mp' =:> s''
 <: end :>
]
```

where s' and s'' may recursively contain placeholders. For every t accepted by the template, that is if t matches mp or mp', the evaluation $start(tmps, t) \Rightarrow s$

is syntax safe, when $s \in \mathcal{L}(G_{intended})$. The match-replace selects either s' or s'', which means that s' and s'' must be a sentence of $\mathcal{L}(G_{intended})$, i.e. a sentence of $\mathcal{L}(A)$ or $\mathcal{L}(A')$. In order to ensure that the template fulfills this requirement the following productions rules are added to $G_{template}$:

- ◇ $S \rightarrow$ "$\langle$: match" *Expr* "$\rangle$" *MatchRuleS* + "$\langle$: end :$\rangle$"
- ◇ *MatchRuleS* $\rightarrow$ "$\langle$: " *MP* " =:$\rangle$" *S*

where *MP* is the nonterminal for the match-pattern syntax and *Expr* for the metalanguage expression syntax. Furthermore the *MatchRule* nonterminal has a suffix *S* to specify that only match-rules containing object code for the nonterminal *S* can be used.

A $G_{template}$ constructed using these rules defines the language of templates resulting in a syntax safe template evaluation for $G_{intended}$. If a template is a sentence of such a $G_{template}$ then its evaluation result is a sentence of $\mathcal{L}(G_{intended})$, since the applied placeholders are always replaced by a (sub)sentence valid for the nonterminal where they are applied. □

After constructing a $G_{template}$ it is possible to build a parse tree of a template. A schematic view of such parse tree of a template is shown in Figure 5.2. G_{object} and $G_{template}$ have the same start symbol. The placeholders are visualized by black sub-parse trees. The G_{meta} part of $G_{template}$ is used to parse them. A special case is the black part that contains a white subtree. At that point a placeholder contains a piece of object code, which is used as a pattern to replace the placeholder, as the case for the match-replace placeholders. The match-replace starts with a piece of meta code and fragments of object code are defined in the match-rules. It is allowed that a white part contains a black part, this black part a white part, and so on. This represents recursive nesting of placeholders, such as a match-replace placeholder applied inside a match rule of another match-replace placeholder.

Having a $G_{template}$ enables the parsing of the entire template. The metalanguage and object language in a template are parsed simultaneously and misspellings in these languages are detected during the parsing phase. Syntax errors are found in the entire template. This helps to find syntax errors in the static part of the template before the template is evaluated, and thus input data is not necessary to detect these errors. A template only based on match-replace placeholders and subtemplates is already syntax safe. When the input data tree is accepted by the template, it generates a sentence of $\mathcal{L}(G_{intended})$. An invalid input data tree will not result in output code, but will result in an error, because it is not accepted by the template.

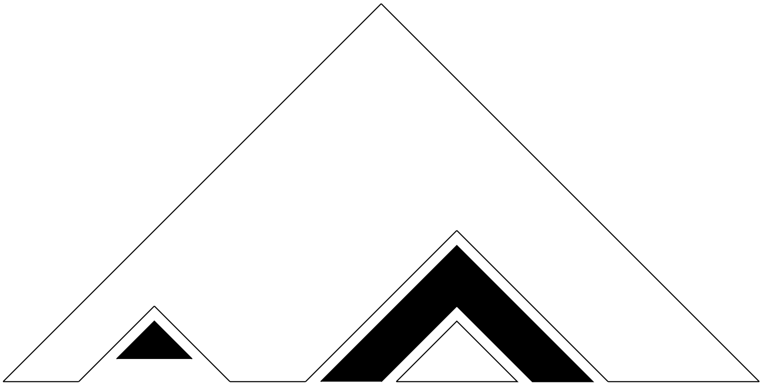


Figure 5.2 Schematic view of a parse tree of a template.

5.3 The Metalanguage Grammar

In this section we present the implementation of template grammars in SDF. The syntax of the placeholders is based on the metalanguage of Chapter 4. The template grammar is obtained by combining the object language grammar and placeholder grammar by adding the placeholder syntax as alternative to the object language nonterminals. Since we use SDF, it is possible to use the *module parameters* to specialize the placeholder syntax for a specific nonterminal, instead of redefining the placeholder syntax for every nonterminal, as proposed by Theorem 5.2.3. An SDF grammar module may have a number of (non)terminal parameters, which can be substituted during the import of a module by the actual required (non)terminal. When the placeholder grammar is added as an alternative for an object language nonterminal, this grammar is parameterized with that object language nonterminal to inject the placeholder syntax as alternative for it. The use of module parameters results in a compact definition of the grammar $G_{template}$. A template grammar is defined by a combination module importing the placeholder syntax for each object language nonterminal which should be extended.

We start with the discussion of syntax shared by all metalanguage constructs. Since some shared constructs can be confusing without knowing specific details of the different constructs, it can be recommended to first read the sections about the different constructs and use the shared syntax as reference. We present the grammars in the order of dependency starting with the shared syntax. After the shared syntax is presented, the syntax of the subtemplates, match-replace placeholder and substitution placeholder is given.

5.3.1 Shared Syntax

The syntax definitions of the three placeholders have a number of shared syntax artifacts. This section discusses the syntax of the hedges, explicit syntactical typing of placeholders and meta-comment.

A. Hedges

Hedges are a syntactical construct used to indicate the transition between object language and metalanguage. In text template systems hedges are obligatory to recognize the placeholders. When a template is parsed and both object language and metalanguage are part of the template grammar, hedges are not longer necessary [115]. The parser will recognize the placeholders as meta code as it cannot be parsed as a piece of object code. If the syntax of the metalanguage overlaps with the syntax of the object language ambiguities can occur. When hedges are used, which are not part of the object language, these ambiguities are efficiently eliminated. In the other cases disambiguation strategies should be defined in order to filter these ambiguities [115].

Besides the ambiguities, the use of hedges makes the transition between meta code and object code visual, so the transition can be easily recognized by humans. Our focus is not on providing templates without hedges and on dealing with the possible ambiguities.

It is essential that the hedges are a sequence of characters disjoint of the syntax of the object language to prevent ambiguities. Since the hedges are only used during the parsing phase and have no semantic meaning, the definition can be overridden by an alternative sequence of characters. We have chosen for the character sequences `<:` and `:>` for aesthetic reasons. Moreover, these character sequences are not commonly used in object languages. These default hedges are defined in the SDF module of Figure 5.3.

B. Syntactical Typing of Placeholders

Template grammars can become ambiguous, since a placeholder can be parsed as multiple nonterminals. In Section 6.7 we discuss the origin of ambiguities in detail and present a filter to handle them. However, the use of an ambiguity filter reduces the performance of the template evaluator. Therefore we included a syntactical construction to force the parser to parse a placeholder as a specified nonterminal. This construction is the `PlaceHolderType[[Sort]]` sort. It can be used to disambiguate a placeholder when multiple types of placeholders fit on a position. The `PlaceHolderType[[Sort]]` sort is explicitly typed via parameterization using the `[[Sort]]` syntax. This is necessary to specialize `PlaceHolderType` for nonterminal `Sort`, which removes overlap between `PlaceHolderType`'s when multiple object language nonterminals are extended.

```
module Common["Sort" Sort]

exports
 sorts BeginTag EndTag

 lexical syntax
   "<:"              -> BeginTag
   ":>"              -> EndTag

   "sort:" "Sort"    -> PlaceHolderType[[Sort]]

   BeginTag "%%" line:~[\n]* EndTag
                     -> LAYOUT
```

Figure 5.3 Syntax shared by all placeholders.

It is, for example, used in Figure 6.8 at line 24 to force that the placeholder `<: $stms sort:STATEMENT* :>` is parsed as the nonterminal `STATEMENT*`.

C. Meta-comment

The last provided syntactical artifact is *meta-comment*: `<:%% meta-comment :>`. Meta-comment is not copied to the output code and its purpose is documenting template code, which is not relevant in the generated code. It is an alternative for the `LAYOUT` sort, which is a special kind of nonterminal in the SGLR implementation in order to handle layout. The `LAYOUT` sort represents all tokens which have no semantic meaning. The other layout artifacts are one-to-one copied to the output, necessary for output languages with layout criteria and to generate human readable output code.

5.3.2 Subtemplates

Two syntactical constructions are necessary for subtemplates. First subtemplates must be declared and identifiable, second a call placeholder is necessary to instantiate a subtemplate. The subtemplate is declared via a label, followed by a fragment of code. The call placeholder for a subtemplate consists of a pair of hedges combined with the identifier of the subtemplate and an expression to provide a new input data context to the subtemplate.

Figure 5.4 shows the grammar definition. The subtemplate is declared by an identifier of the sort `IdCon` followed by the object code of syntactical type `Sort` between square brackets. The call placeholder consists of a couple of hedges, an identifier and an expression. The syntax of the hedges and the expressions are provided via imported grammars. The expression grammar defines the syntax for the meta-variables, tree queries and string concatenation. Optionally it can be disambiguated using the syntax defined by `PlaceHolderType[[Sort]]`.

```
module PlaceHolderSubTemplate["Sort" Sort]

imports Common["Sort" Sort]
imports basic/IdentifierCon
imports Expression

exports
  sorts Template

  context-free syntax
    IdCon "[" Sort "]" -> Template

    BeginTag IdCon"(" Expression ")"
             PlaceHolderType[[Sort]]? EndTag
                        -> Sort {placeholder("subtemplate")}
```

Figure 5.4 Syntax for subtemplates.

The annotation `placeholder("subtemplate")` is used by the evaluator, see Chapter 6, to detect the placeholder.

In order to parse a template with call placeholders, it is necessary to embed the placeholder syntax in the object language. This grammar module defines the placeholder syntax with the generic result nonterminal `Sort`. The definition can be added as an alternative to an arbitrary object language nonterminal. The injection of the placeholder in the object language is achieved by importing the placeholder grammar, while parameterizing this module with an object language nonterminal. During parameterization `Sort` is internally replaced by the object language nonterminal, as a result the placeholder syntax becomes an alternative for that object language nonterminal.

The parameterizing of `Sort` is compliant with the proposed construction of template grammars in Theorem 5.2.3. The identifier nonterminal used to label the subtemplates generalizes from the fixed label in production rules supposed by Theorem 5.2.3. This means the template evaluator should check whether the called subtemplate result in the correct grammatical sort. Syntax safe evaluation is discussed in Chapter 6.

An example of the use of a subtemplate can be found in Figure 6.8. At lines 39-50 the subtemplate `expr` is declared. This subtemplate is for instance called at line 23.

5.3.3 Match-Replace

The match-replace placeholder is a construction containing multiple match rules. Each match rule has a fragment of object code accompanied by a tree match-pattern. The match-replace replaces itself with a fragment of object code defined in the match rule. This implies that the fragment of object code in a

```
module PlaceHolderMatchReplace["Sort" Sort]

imports Common["Sort" Sort]
imports Expression
imports MatchPattern

exports
  sorts MatchRule
  context-free syntax
    BeginTag "match" Expression
             PlaceHolderType[[Sort]]? EndTag
        MatchRule[[Sort]]+
        BeginTag "end" EndTag
             -> Sort {placeholder("matchreplace")}

    BeginTag MatchPattern "=" EndTag
        Sort
             -> MatchRule[[Sort]]
```

Figure 5.5 Syntax for match-replace placeholder.

match rule must be parsed as the same object language nonterminal as the match-replace substitutes.

The principle of extending the object language is the same as the subtemplate call placeholder. The match-replace placeholder is also injected as an alternative for an object language placeholder. Furthermore, it supports an expression to specify the context for the match-rules. This expression is evaluated before the the match-rules are tried. Figure 5.5 shows the generic match-replace grammar module.

A match rule contains a `MatchPattern` and a fragment of resulting object language code. The `MatchPattern` is a tree pattern containing possible meta-variables. The syntax depends on the particular tree representation. In our case ATerms [22] are used, but it can also be another tree syntax like XML or JSON representation. Beside the syntax for the tree representation, the `MatchPattern` supports meta-variables defined as a dollar sign followed by a label with the character class `[A-Za-z][A-Za-z\-0-9]*`.

The match-replace syntax definition is also compliant with the proposed construction of template grammars in Theorem 5.2.3. The result code of a match rule is of the same syntactical type as the injection of the match-replace, which is visible in the grammar definition. The `MatchRule` contains the `Sort` nonterminal and finally the match-replace is injected in the `Sort` nonterminal as alternative. The following chain of productions is recognizable: $\mathit{Sort} \Rightarrow \mathit{MatchRule}[[\mathit{Sort}]]+ \Rightarrow \mathit{Sort}$. This structure enforces that all possible results of match-replace are of the nonterminal `Sort`, i.e. it is not possible to mix different sorts in the match rule set. The `MatchRule` nonterminal is augmented with the parameterized name `[[Sort]]`, so that the `MatchRule` nonterminal is unique for every object nonterminal extended with placeholder syntax.

```
module PlaceHolderSubstitution["Sort" Sort]

imports Common["Sort" Sort]
imports Expression

exports
  context-free start-symbols Sort
  context-free syntax
    BeginTag Expression PlaceHolderType[[Sort]]? EndTag
                      -> Sort {placeholder("substitution")}
    "end"             -> IdCon {reject}
```

Figure 5.6 Syntax for substitution placeholder.

An example of the use of a match-replace placeholder can be found in Figure 6.8. At lines 5-14 a match-replace placeholder is used.

5.3.4 Substitution Placeholder

The discussion of the syntax of the substitution placeholder may sound odd, since it can be expressed using subtemplates and match-replace placeholders. There are two reasons to discuss this placeholder. First, the substitution is an intuitive construct, which allows substituting a string of the input data into a template. Second, in a syntax safe template evaluator we can add implicit behavior to it, which we will discuss in Chapter 6. The difference with the previous two discussed metalanguage constructs is that syntax safety is not enforced by the grammar, but must be handled by the template evaluator.

In short a substitution placeholder consists of a couple of hedges and an expression to obtain the data to replace it. The syntactical pattern for the substitution placeholder is `<: Expr :>` or `<: Expr sort:SORT :>`, where the capitalized `SORT` is the nonterminal name. This last construction is used to force the parser to parse the placeholder as the given syntactical type to solve ambiguities. A generic syntax definition of the substitution placeholder is given in Figure 5.6.

The substitution placeholder grammar module declares the `Sort` as start symbol. This grammar is also used by the template evaluator to check if the string provided by the input data is allowed to replace the substitution placeholder.

A final note: For aesthetic and readability reasons we have chosen for the word `end` in the closing tag of the match-replace placeholder. Since `end` can be parsed as nonterminal `IdCon` an ambiguity may occur on the closing tag of the match-replace placeholder given that it also can be recognized as a substitution placeholder. The `end` label is rejected as `IdCon` in order to remove the chance of this ambiguity. This reject restricts the use of the word `end` as label in the

```
module PlaceHolder["Sort" Sort]

imports PlaceHolderSubstitution["Sort" Sort]
imports PlaceHolderSubTemplate["Sort" Sort]
imports PlaceHolderMatchReplace["Sort" Sort]
```

Figure 5.7 Combination of all placeholders to a single module.

input data. One can choose to replace `end` with a sequence of characters which cannot be parsed as an `IdCon` in order to remove this restriction.

5.4 Grammar Merging

We have discussed the generic parts of the metalanguage grammar. The template grammar $G_{template}$ is instantiated by combining the G_{object} and G_{meta} via importing both grammars in a combination module and describing their connection. In order to connect the object language and the metalanguage, nonterminals of the G_{object} are extended with an extra alternative to connect the object language nonterminal with the root nonterminal of the metalanguage.

The presented placeholder syntax definitions are independent of the object language grammar. In order to inject the placeholder syntax in the object language grammar, the placeholder grammar modules are imported in the combination module and at the same time parameterized with a nonterminal of the object language grammar. At the moment that the placeholders are imported and parameterized with a nonterminal of the object language, the placeholder is injected as an alternative for that nonterminal. Since all three placeholder construct are simultaneously applied to a nonterminal, the three previously defined placeholder modules are combined in a single module called `PlaceHolder`, which is shown in Figure 5.7. A small note must be made on the parameterization arguments: `"Sort"` must be the literal name of the `Sort` and is used as a keyword to fix the sort of substitution placeholders syntactically, since SDF has no syntax to do this automatically.

The last grammar module, shown in Figure 5.8, is of a more operational nature. It defines a start symbol `TemplateSet` representing a list of (sub)templates. The template with the identifier `template` is the starting point of the evaluator and it contains object code for `Sort`, which is most likely the start symbol of the object language. This start template may be accompanied by a file name to instantiate a file containing the generated code. This is a requirement for using templates for case studies and other real world applications. The nonterminals `Filename` and `SubDirectory` belonging to the nonterminal `File` are extended with placeholders to enable parameterization. The `Template*` nonterminal is

```
module StartSymbol["Sort" Sort]

imports basic/IdentifierCon
imports utilities/fileIO/Directory
imports PlaceHolderSubstitution["Filename" Filename]
imports PlaceHolderSubstitution["SubDirectory" SubDirectory]
imports PlaceHolder["Template*" Template*]

exports
  context-free start-symbols
    TemplateSet

  sorts Template

  context-free syntax
    "[" Template* "]"                    -> TemplateSet
    "template" "[" File "," Sort "]"     -> Template
    "template" "[" Sort "]"              -> Template
    "template"                           -> IdCon {reject}
```

Figure 5.8 Start symbol module.

also extended with placeholder syntax to enable iteration. Figure 5.8 shows the start symbol module.

All ingredients for the combination module to instantiate a template grammar are defined. The combination module is the only object language specific part of the $G_{template}$ definition. It imports the object language, it instantiates the `StartSymbol` module and it defines which nonterminals of the object language are injected with placeholder syntax.

A side effect of the injection of placeholders in an object language is that the grammar $G_{template}$ often becomes highly ambiguous. The ambiguities are caused by the possibility to recognize multiple parameterized placeholders. The chance of ambiguities increases if more object language nonterminals are extended with placeholder syntax. Therefore automatic parameterization of placeholders with every object language nonterminal is undesired. The selection process of the nonterminals for parameterization of the placeholders must be done manually. In Section 6.7 we will discuss this problem in more detail. It is hard to automatically predict which sorts must be selected for the parameterization of placeholders. A similar problem is discussed in [119]: although the authors generate their connection rules, they consider it useful to have full control over the selection of the nonterminals.

Figure 5.9 shows the combination module for the PICO Language. This template grammar can parse the unparser template of Figure 4.7.

There is one requirement left for the object language grammar. Nonterminals in G_{object} must not be defined in G_{meta}, that is $N_{object} \cap N_{meta} = \emptyset$; otherwise undesirable and uncontrolled nonterminal injections occur. In practice this can be simply achieved by adding a unique prefix or suffix to the (non)terminals of

```
module Template-Pico

imports Pico

imports StartSymbol["PROGRAM*" PROGRAM*]
imports PlaceHolder["NatCon" NatCon]
imports PlaceHolder["StrCon" StrCon]
imports PlaceHolder["TYPE" TYPE]
imports PlaceHolder["PICO-ID" PICO-ID]
imports PlaceHolder["EXP" EXP]
imports PlaceHolder["STATEMENT" STATEMENT]
imports PlaceHolder["ID-TYPE" ID-TYPE]
imports PlaceHolder["DECLS" DECLS]
imports PlaceHolder["PROGRAM" PROGRAM]
imports PlaceHolder["STATEMENT*" {STATEMENT ";" }*]
imports PlaceHolder["ID-TYPE*" {ID-TYPE "," }*]
```

Figure 5.9 PICO combination module.

a grammar, or a typical SDF specific solution consists of parameterization of all nonterminals of a language with its language name to create a namespace [27].

The modularity of SDF allows us to specify the metalanguage and object language grammars separately. The advantage of this approach is the ease of using off-the-shelf object language grammars [70]. In case an off-the-shelf object language grammar is not available and full syntax checking of templates is not required, one can decide to use an *island grammar* [82]. An island grammar only defines small parts of a language. The rest of the language is defined at a global level, for example as a list of characters.

The presented grammar modules are defined in SDF. It should be possible to use another syntax formalism. When modularization and parameterization are not available one can choose to instantiate the placeholder injections using a code generator. The most important feature is that the chosen parser supports ambiguities, but this requirement does not depend on the chosen syntax formalism.

5.5 Related Work

We discuss SafeGen, a template like code generation approach guaranteeing at least syntax safety. SafeGen [61] is an approach aiming for type safe templates. It uses an automatic theorem prover to prove the well-formedness of the generated code for all possible inputs.

This approach heavily depends on the assumptions that the input is a valid Java program and the knowledge of the Java type system. The template programmer can define placeholders (cursors) to obtain data from the Java input program. Those placeholders must contain constraints based on their use in the template.

For example a placeholder in the `extends` section of a class in a template must guarantee the extended class is not final. A prover used to check the constraints ensures that the template cannot generate ill-formed code.

SafeGen depends on the knowledge that the input and output program is Java. This fact makes the environment incapable of generating code from an abstract high-level input data in another representation than the object language, because the approach depends heavily on reflection. Although the approach could give more and better guarantees about the generator, switching to another object language and input data representation is hard. A template grammar, such as described here, is more flexible in the choice of input data language and output language.

Another approach to achieve syntax safe code generation is using *abstract parsing* [75]. Abstract parsing is a statical analysis technique for checking the syntax of generated strings. It uses data-flow analysis and checks via a kind of parser if the data-flow produces a sentence conform the intended output language grammar. During runtime no further checking is required. This technique requires an analyzer for the data-flow to feed the parser, while we directly parse the template. Abstract parsing is a generalization of the template grammar we presented, as it can be used for every kind of meta program, when data-flow analysis is possible. In contrast with abstract parsing, our approach does not need external data-flow analysis to achieve syntax safety.

5.6 Conclusions

In this chapter we have presented a grammar to parse all languages in a template simultaneously. Syntax errors in the object code of a template are detected while parsing the template, instead of dealing with syntax errors at compile time of the generated code. The whole template is parsed, and thus checked for syntax errors, including rarely generated code in conditional placeholders. Checking a template offers accurate error messages, instead of checking the generated code such as the output of the C preprocessor [41]. It is the first step to achieve more safety in template evaluation and helps to avoid syntax errors, like misspellings. The templates are syntactically not different from text templates and as a result they provide the same user experience.

The modular grammar definition combined with parameterization allows to instantiate template grammars for different object languages with minimal redefinition and cloning. The advantage of this approach is the ease of using off-the-shelf object language grammars [70].

Having the parse tree of both object language and metalanguage also creates the possibility to implement advanced IDE[2] features. Templates are not well

[2] Integrated Development Environment

supported by IDE's due to their multilingual nature. Syntax highlighting, which we already get for free from the ASF+SDF meta-environment [21], and other source code based features for both languages are possible.

Parsing templates on its own is not sufficient to guarantee that the output of the template evaluator is a sentence of the output language. In Chapter 6 we will discuss syntax safe evaluation and the requirements for it.

6

Repleo: Syntax Safe Template Evaluation

Parsing a template is not sufficient to achieve syntax safe code generation. The template evaluator must guarantee that it instantiates a parse tree belonging to the set of parse trees of the object language. This guarantee is achieved by checking that the root nonterminal of the sub parse tree replacing a placeholder is equal to the object language nonterminal where the placeholder is applied. Syntax safe evaluation also allows implicit subtemplates in order to express list rendering and tree rendering in a more compact fashion. Furthermore since templates can easily become ambiguous, an ambiguity filter is discussed. Finally, a number of case studies show that syntax safe templates can be used for industrial programming languages.

6.1 Introduction

Parsing a template alone is not sufficient to achieve syntax safe code generation. By introduction of the substitution placeholder and the free choice of identifiers for subtemplates, correctness of the parse tree of a template does not imply that the generated code will be syntactically correct. For example, a subtemplate call placeholder with identifier s is applied for nonterminal n_1, while the root nonterminal of the subtemplate with the identifier s is n_2 and thus not equal to n_1. Although the substitution placeholder breaks static verification of syntax safety, it is a design choice to leave it in our metalanguage as it is one of the common constructs in a template metalanguage. The substitution placeholder is offered by most (industrial) template evaluators. Examples of applications of the substitution placeholder in other template evaluators are shown by Figure 1.12, Figure 4.10, Figure 4.12 and Figure 4.13. The substitution placeholder allows replacing it by unstructured strings stored in the input data.

Therefore, it is necessary to dynamically verify that the string stored in the input data is allowed to replace the placeholder.

This chapter discusses the implementation of a syntax safe template evaluator. The template grammar is used to obtain a parse tree of a template and to check the syntax of object code and meta code in one parse phase. The template evaluator uses the object language grammar while substituting the placeholders to guarantee that the output parse tree complies to the object language grammar. Syntax safe evaluation is achieved by checking that a placeholder parsed as nonterminal A_{object} is replaced by a sub parse tree where the root is nonterminal A_{object}. We have implemented our approach in a prototype called *Repleo* to facilitate empirical validation of syntax safe evaluation (see Chapter 7).

We start with a discussion of the syntax safe evaluation function, followed by the specific evaluation for the substitution placeholder, match-replace placeholder and subtemplate call placeholder in Sections 6.3, 6.4 and 6.5. We introduce the implicit subtemplate mechanism and discuss the revisited substitution placeholder evaluation in Section 6.6. In addition to the discussion of the placeholders, the evaluator also has to resolve ambiguities and handle separators in Section 6.7 and 6.8. After the introduction of syntax safe evaluation, we show some examples of syntax safe templates including a reimplementation of the PICO unparser using the additional semantic properties of the metalanguage in Section 6.10.

6.2 Syntax Safe Evaluation

The operational semantics of syntax safe evaluation are based on the semantics of the metalanguage discussed in Section 4.3. We extend these operational semantics since the information in the parse tree can be used to express implicit subtemplates, resulting in more concise templates.

The core of the syntax safe evaluator is the same single tree traversal as discussed in Section 4.3, with a slightly different signature:

$$seval : Template \times Templates \times MVars \rightarrow Tree,$$

where *Template* is the current template parse tree, *Templates* is a symbol table containing (sub)template parse trees, *MVars* is a symbol table containing metavariables and *Tree* is the parse tree resulting from the template evaluation. The functions is called *seval* to emphasize the difference between the non syntax safe template evaluator *eval* function. The *seval* function differs from the *eval* function of Section 4.3 in the following respects:

- ◇ The output is a parse tree of the output language, the *yield* function is used to convert it to a string;

◇ *Template* and *Templates* are parse trees rather than strings; lexical analysis is performed during the template parsing;

◇ The symbol table *Templates* is a block-structured symbol table (see Section 4.3.2).

The *seval* function traverses the parse tree and checks whether the current node is a placeholder. At the moment a placeholder is found, the type of placeholder (substitution, match-replace or subtemplate call) is obtained and the accompanying *seval* evaluation sub function is invoked. The result of that sub function is a sub parse tree replacing the placeholder node, unless an error occurs. The final result of the evaluator function is a parse tree without placeholders, or an error message.

In Section 5.3 we presented the grammars for the placeholders. The production rules for the placeholders are extended by the annotation `placeholder(...)`, containing the kind of the placeholder, i.e. `substitution`, `subtemplate`, or `matchreplace`, which is used by *seval* to select the evaluation function. This annotation is an implementation choice simplifying placeholder detection in the parse tree. The traversal function matches on the annotation instead of the syntactical pattern of the placeholder.

When *seval* detects a placeholder in the parse tree, it is necessary to know to what object language nonterminal it is applied, i.e. the type of the parent node in the parse tree. Therefore, to obtain the object language nonterminal of the placeholder we introduce a helper function $getparentnt : Tree \rightarrow A$. There are several ways to implement this function. For example, *seval* can be extended by an extra parameter holding the parent nonterminal, or the nonterminal of the placeholder can be parameterized by the parent nonterminal. We use a specific technical solution based on the assumption that the SGLR parser implementation is used. The SGLR parser implementation produces verbose parse trees, where every node is augmented with the production rule used for instantiating it. The *getparentnt* function uses this production rule to obtain the parent nonterminal. It returns the producing nonterminal from the production rule, which is equal to the parent node in the parse tree.

6.3 Substitution Placeholder

The presence of the substitution placeholder results in the biggest difference between the *eval* function and the *seval* function. The substitution placeholder allows one to have unstructured data in the form of a string in the input data tree and to use it for replacing a substitution placeholder. A mechanism is necessary to verify that it is valid to substitute the placeholder in the template with the string from the input data.

A substitution placeholder of nonterminal A_{object} can be safely replaced by a sub parse tree with root A_{object}. The expression specified in the substitution placeholder can result in a string or a tree. In Section 6.6 we discuss the behavior when the result of the expression evaluation is a tree. In case the expression evaluator yields a string, it is necessary to convert the string to the corresponding parse tree and check whether its root is A_{object}. We implemented the check using a parser for the object language. When the parsing succeeds and the root nonterminal of the (sub) parse tree is A_{object}, the parse tree can safely substitute the placeholder. In case the string of the input data cannot be parsed or the root nonterminal differs from A_{object}, i.e. the string is not a sentence of $\mathcal{L}(G(A_{object}))$, an error message is generated.

The following equation specifies the behavior of the syntax safe substitution evaluation. For ease of presentation we use concrete object syntax notation in the equation. Inspired by [119], the notation $[[\ldots]]$ is used to specify concrete syntax, which is internally represented as a parse tree.

$$\frac{\begin{array}{c} getparentnt([[<: expr :>]]) \mapsto A \\ evalexpr([[expr]], bst_{vars}) \mapsto s \\ parse_{G(A)}(s) \mapsto t \end{array}}{seval([[<: expr :>]], bst_{tmps}, bst_{vars}) \mapsto t}$$

$$\frac{\begin{array}{c} getparentnt([[<: expr :>]]) \mapsto A \\ evalexpr([[expr]], bst_{vars}) \mapsto s \\ parse_{G(A)}(s) = ERROR \end{array}}{seval([[<: expr :>]], bst_{tmps}, bst_{vars}) \mapsto ERROR}$$

6.4 Match-replace Placeholder

The evaluation of match-replace placeholders does not differ from the evaluation scheme discussed in Section 4.3.4. Only at the start of the evaluation of the match-replace, the (sub) parse tree belonging to the match-replace is added to the (sub)templates symbol table in a fresh scope. The label for the subtemplate is equal to the nonterminal of the match-replace prefixed with a _. The prefix _ is not allowed for manual declared (sub)templates, so the labels of the implicit placeholders are hygienic with respect to the labels of manually declared (sub)templates.

The use of scopes is necessary to remove conflicts if multiple implicit subtemplates are added to the symbol table with the same name, i.e. the same nonterminal of the match-replace. Furthermore, the scoping mechanism facilitates that the last added implicit subtemplate is selected, when multiple match-replace placeholders for the same nonterminal are nested.

The formalized behavior specification is presented below. It is almost equivalent to the equation of Section 4.3.4, except that it uses parse trees instead of strings. Furthermore, the sub parse tree of the match-replace is stored on the subtemplate stack.

$$\frac{\begin{array}{c} t_1 = [[\text{<: match :>}[mr_1, \ldots, mr_i] \text{ <: end :>}]] \\ getparentnt(t_1) \mapsto A \\ startblk(bst_{vars1}) \mapsto bst_{vars2} \\ startblk(bst_{tmps1}) \mapsto bst_{tmps2} \\ add(bst_{tmps2}, \_A, t_1) \mapsto bst_{tmps3} \\ lookup(bst_{vars2}, \$\$) \mapsto t_2 \\ findmatch(t_2, [mr_1, \ldots, mr_i], bst_{vars2}) \mapsto \langle t_3, bst_{vars3} \rangle \\ seval(t_3, bst_{tmps3}, bst_{vars3}) \mapsto t_4 \end{array}}{seval([[\text{<: match :>}[mr_1, \ldots, mr_i] \text{ <: end :>}]], bst_{tmps1}, bst_{vars1}) \mapsto t_4}$$

6.5 Subtemplate Placeholder

The behavior of the subtemplate call placeholder is that it should replace itself with a sub parse tree with root nonterminal A_{object} which is equal to the A_{object} where the call placeholder is applied. The behavior of the subtemplate call placeholder is not changed with respect to the operation definition in Section 4.3.3. A subtemplate is selected based on its identifier. However, the free choice of an identifier for subtemplates results in the risk that the subtemplate is not of the correct syntactic sort to replace the subtemplate call placeholder.

In order to guarantee syntax safety, the subtemplate call placeholder must be replaced by a sub parse tree with the correct root nonterminal. Hence, we are only allowed to insert the result of an evaluated subtemplate when its root nonterminal matches the nonterminal where the subtemplate call placeholder is applied. An extra condition is added to the original operation semantics to check whether the root nonterminal of the subtemplate matches the nonterminal of the calling placeholder. An error is generated in case no suitable subtemplate can be found.

$$\frac{\begin{array}{c} getparentnt([[\text{<: } idcon(expr) \text{ :>}]]) \mapsto A_1 \\ lookup(bst_{tmps}, idcon) \mapsto t \\ getparentnt(t) \mapsto A_2 \\ A_1 = A_2 \\ evalexpr([[expr]], bst_{vars1}) \mapsto bst_{vars2} \\ seval(t, bst_{tmps}, bst_{vars2}) \mapsto t' \end{array}}{seval([[\text{<: } idcon(expr) \text{ :>}]], bst_{tmps}, bst_{vars1}) \mapsto t'}$$

$$\frac{\begin{array}{c} \mathit{getparentnt}([[<:\ \mathit{idcon}(\mathit{expr})\ :>]]) \mapsto A_1 \\ \mathit{lookup}(\mathit{bst}_{\mathit{tmps}}, \mathit{idcon}) \mapsto t \\ \mathit{getparentnt}(t) \mapsto A_2 \\ A_1 \neq A_2 \end{array}}{\mathit{seval}([[<:\ \mathit{idcon}(\mathit{expr})\ :>]], \mathit{bst}_{\mathit{tmps}}, \mathit{bst}_{\mathit{vars}}) \mapsto \mathit{ERROR}}$$

6.6 Substitution Placeholder Revisited

In the discussion of the metalanguage in Chapter 4 we introduced the concept of subtemplates to enable a kind of unfolding in order to render lists and trees, or to reduce clones in the templates. The disadvantage of using the match-replace and subtemplates to express the rendering of lists and trees is that it is quite verbose. Furthermore, the structure of the generated code is less clear, since a subtemplate placeholder is called at the place where the list or tree must be rendered. The availability of syntax safe templates enables us to define implicit subtemplates, providing a more natural way to express unfolding. It uses the syntactical type of the match-replace placeholder to automatically add an implicit subtemplate to the subtemplates symbol table and the syntactical type of the substitution placeholder to invoke the subtemplate.

The concept of implicit subtemplates is shown by the following figures. Figure 6.1 shows a snippet of the original PICO unparser with a subtemplate `decls`. This subtemplate can be re-factored to an implicit subtemplate. Figure 6.2 shows the integration of the subtemplate `decls` in the first match-replace placeholder. The subtemplate call `<: decls($tail) :>` is replaced by a substitution placeholder `<: $tail :>`. Since the match-replace placeholder will be parsed as `Decls` and the placeholder `<: $tail :>` will also be parsed as `Decls`, this mechanism will render a list of declarations. We can also omit the `[ head ]` match rule as specified in the original PICO unparser of Chapter 4, since the syntax safe evaluator provides a generic separator handler (see Section 6.8).

Implicit subtemplates use the property that the object language nonterminal can be used as a label for a subtemplate. As mentioned in Section 6.4, the sub parse tree of a match-replace placeholder is added to symbol table containing the subtemplates. The match-replace placeholder acts as an implicit subtemplate, where the identifier is the object language nonterminal prefixed by an underscore. The implicit subtemplate can be called by a substitution placeholder in the object code fragments of the match rules. A substitution placeholder becomes an implicit subtemplate caller, when its expression results in a subtree of the input data instead of a string. At the moment the expression returns a tree, an implicit subtemplate is selected based on the syntactical type of the substitution placeholder. The last added implicit subtemplate with the same syntactical type as the substitution placeholder is used for evaluation. The current input data context is set to the subtree selected by the expression of

```
template[
<: match :>
 <: program( decls($decls), $stms ) =:>
  begin declare
   <: decls( $decls ) :>;
   <: stms( $stms ):>
  end
<: end :>
]

decls[
<: match :>
   <: [] =:>
   <: [ $head ] =:>  <: idtype($head) :>
   <: [ $head, $tail ] =:> <: idtype($head) :>, <: decls($tail) :>
<: end :>
]
```

Figure 6.1 Original snippet of PICO abstract syntax tree unparser.

```
template[
<: match :>
 <: program( decls($decls), $stms ) =:>
  begin declare
   <: match :>
    <: [] =:>
    <: [ $head, $tail ] =:> <: idtype($head) :>, <: $tail :>
   <: end :>;
   <: stms( $stms ):>
  end
<: end :>
]
```

Figure 6.2 Implicit subtemplate example.

the substitution placeholder. If no implicit subtemplate can be found, an error will be generated. The additional rules for the behavior of the substitution placeholder calling an implicit placeholder are shown below.

$$\frac{\begin{gathered} getparentnt([[<:expr:>]]) \mapsto A \\ evalexpr([[expr]], bst_{vars1}) \mapsto t \\ startblk(bst_{vars1}) \mapsto bst_{vars2} \\ add(bst_{vars2}, \$\$, t) \mapsto bst_{vars3} \\ lookup(bst_{tmps}, \_A) \mapsto t_1 \\ seval(t_1, bst_{tmps}, bst_{vars3}) \mapsto t_2 \end{gathered}}{seval([[<:expr:>]], bst_{tmps}, bst_{vars1}) \mapsto t_2}$$

$$\frac{\begin{array}{c} getparentnt([[<: expr :>]]) \mapsto A \\ evalexpr([[expr]], bst_{vars1}) \mapsto t \\ startblk(bst_{vars1}) \mapsto bst_{vars2} \\ add(bst_{vars2}, \$\$, t) \mapsto bst_{vars3} \\ lookup(bst_{tmps}, \_A) = \epsilon \end{array}}{seval([[<: expr :>]], bst_{tmps}, bst_{vars1}) \mapsto ERROR}$$

6.7 Ambiguity Handling

Adding placeholders to the grammar of an object language to obtain a template grammar can lead to undesired ambiguities. An ambiguity occurs when a (sub) parse tree of a (sub) sentence can be constructed in multiple ways using the production rules of a context-free grammar. In the situation of the constructed template grammars ambiguities can have two causes: either the object language grammar itself is already ambiguous or the combination of object language grammar and metalanguage grammar introduces ambiguities. Ambiguities are a problem when parsing (source) code, because it can lead to misinterpretation. Although those ambiguities are unwanted when analyzing source code, they do not matter when generating source code, as we will show in this section.

Instantiating template grammars, by adding placeholders to an object language grammar, can introduce new ambiguities. Placeholders are added as alternative for different object language nonterminals, but the syntax of these placeholders is equal. The parser cannot distinguish syntactically the desired derivation when different placeholders fit, hence the introduction of the explicit syntactical typing for placeholders via the optional syntax of the nonterminal `PlaceHolderType[[Sort]]`. There are two causes for ambiguities resulting by adding placeholders as alternatives for object language nonterminals:

- ◇ Multiple sibling alternative placeholders can be used to parse the sentence;
- ◇ The placeholders are defined in chain rules and multiple chain rules can be used to parse the sentence.

The following examples illustrate these two causes. The first example is based on different possible sibling alternatives for a nonterminal. This kind of ambiguity is introduced when a grammar has the following two production rules: $n_1 \rightarrow n_3, n_2 \rightarrow n_3$, where both n_1 and n_2 are extended with a placeholder. Consider the grammar of Figure 6.3. The grammar becomes ambiguous when placeholder syntax is added for nonterminals `A` and `B`. Figure 6.4 shows the ambiguous parse tree of the template `<: v1 :>`. The placeholder can be either parsed as child of the nonterminal `A` or as child of the nonterminal `B`. It depends on the value stored in the input data for the node `v`, whether the final

```
context-free start-symbols C
context-free syntax
  A | B        -> C
  "a"          -> A
  "b"          -> B
```

Figure 6.3 Grammar with two alternatives.

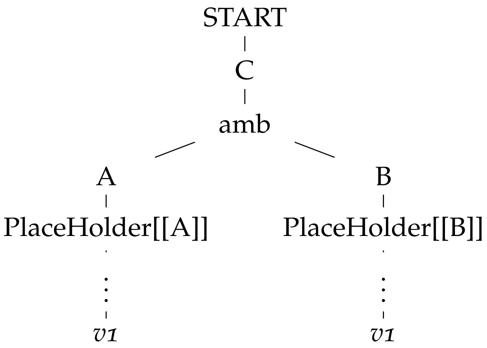


Figure 6.4 Parse tree of ambiguous template.

evaluated template results in the branch for nonterminal `A` or the branch for nonterminal `B`.

During evaluation, the final selection of which alternative succeeds depends on the content of the input data. Consider the following two inputs:

1. `v( "a" )`
2. `v( "b" )`

Both inputs will result in a valid parse tree for the object language. The first input yields the left branch of the ambiguity, the second input the right branch.

The second example of an ambiguity is based on chain rules. The second cause for ambiguities is the presence of chain rules in the grammar, where parent and child nonterminal both are extended with a placeholder. A production rule is a *chain rule*, when it has the following form $n_1 \to n_2$. Consider the grammar of Figure 6.5 and define placeholders for the nonterminals `A` and `B`; the grammar becomes ambiguous. Figure 6.6 shows this ambiguity inside the parse tree of the term `<: "a" :>`. The placeholder can parsed either as an placeholder for the nonterminal `A` or `B`. Considering Figure 6.5, after evaluation `PlaceHolder[[A]]` will return a sub parse tree with `A` as root and `"a"` as child, while evaluating `PlaceHolder[[B]]` will return a sub parse tree with `B` as root,

```
context-free start-symbols B
context-free syntax
  A             -> B
   "a"          -> A
```

Figure 6.5 Grammar with chain rules.

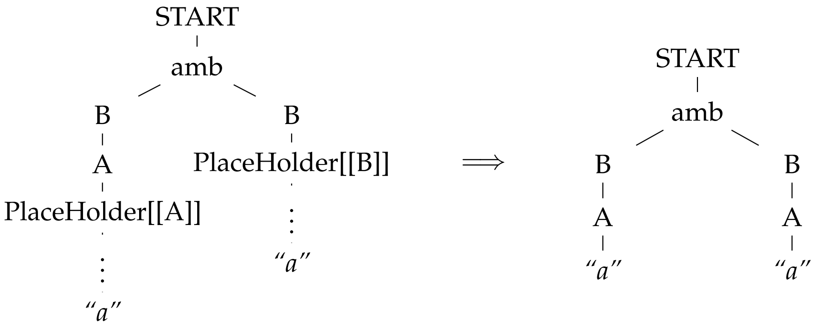


Figure 6.6 Chain rule ambiguity and evaluation result.

`A` as child of `B` and `"a"` as terminal. When both branches of the ambiguity are evaluated, both branches of the ambiguity will contain the same sub parse tree.

A parser used to parse a template should be based on an algorithm supporting ambiguities, as by nature a template grammar is probably ambiguous. The support of ambiguities is the reason for the use of a(n) (S)GLR based parser [117], which constructs a collection of (sub) parse trees in case of an ambiguity. The SGLR parser automatically constructs a parse tree, where a special `amb` node is introduced in case of an ambiguity. This `amb` node contains the list of possible valid (sub) parse trees. Other parser algorithms supporting ambiguities such as the Cocke-Younger-Kasami algorithm [88] can be used, parser algorithms like LL, LR and LALR [5] cannot be used as they do not support ambiguities. Ambiguities belonging to the mix of object language and metalanguage must not be solved during parsing, but stored in the parse tree. These ambiguities represent different syntactically legal alternatives for the output code. It is undesired to remove legal alternatives during parsing if not explicitly defined in the templates.

We use a disambiguation filter based on rewriting [114] to resolve the ambiguities during template evaluation. In order to deal with ambiguities the evaluator tries to evaluate the different alternatives of the ambiguities with the same context, i.e. bst_{tmps} and bst_{vars}. The alternatives are evaluated one by one and at the moment an alternative successfully evaluates, it is used to replace the ambiguity node. The steps are shown by the equations below. This filter is part of the *seval* traversal function and matches on ambiguity nodes.

$$\frac{seval(t_1, bst_{tmps}, bst_{vars}) \mapsto t}{seval(amb(t_1, \ldots, t_k), bst_{tmps}, bst_{vars}) \mapsto t}$$

$$\frac{\begin{array}{c} seval(t_1, bst_{tmps}, bst_{vars}) = ERROR \\ seval(amb(t_2, \ldots, t_k)) \mapsto t \end{array}}{seval(amb(t_1, t_2, \ldots, t_k), bst_{tmps}, bst_{vars}) \mapsto t}$$

$$\frac{seval(t_1, bst_{tmps}, bst_{vars}) = ERROR}{seval(amb(t_1), bst_{tmps}, bst_{vars}) \mapsto ERROR}$$

It is not necessary to evaluate every alternative of the ambiguity when a successful evaluation is found [25]. Evaluating different ambiguity alternatives could result in different structures of the parse trees, but the leaves of those trees contain the same lexical characters. The yielded strings of different ambiguity alternatives are identical. This fact and the fact that after evaluation the resulting parse tree is converted to a string, allows to stop evaluating the ambiguity alternatives after one successful result is found. The evaluator generates an error when it is not possible to evaluate any of the ambiguity alternatives successfully.

Although ambiguities could not always be avoided during the construction of a template grammar, one should carefully select the object language nonterminals to extend with placeholders syntax to prevent undesired ambiguities. The remaining template related ambiguities can be resolved by the algorithm presented above, but this solution comes with a performance penalty.

6.8 Separator Handling

By design choice, our syntax safe templates are based on SDF [55], a grammar formalism supporting lists with separators. When a template is evaluated containing separated lists, it is not automatically guaranteed that the separated lists in the generated code are syntactical correct. One can write a syntactically correct template, which generates syntactically incorrect code. Figure 6.7 shows an example which generates code with a syntax error. The comma at the second match-rule is parsed as separator. Following the evaluation rules of the match-replace, this template will generate a terminated list $(t, t, t,)$ instead of a separated list, while a separated list (t, t, t) is required. In a text-based context one can solve this problem by introducing a third rule between line

```
1 <: match :>
2  <: [] =:>
3  <: [ $head, $tail ] =:> <: $head :>, <: $tail :>
4 <: end :>
```

Figure 6.7 Separated list generation.

2 and 3: `<: [ $head ] =:> <: $head $>`. This solution is also chosen in the text-based PICO unparser of Figure 4.7.

In case of syntax safe templates based on SDF, where a verbose parse tree is available this fix is not necessary. The evaluator uses the information in the parse tree to fix the separators and guarantee that the separator in a separated list is correctly applied. The filter checks whether a separated list conforms to the pattern $n_1 s n_2 \dots s n_k$, where n represents an element and s the separator. It adds or deletes separators in case they are missing or redundant. The same kind of solution is also used in the ASF evaluator [13].

6.9 Repleo

In order to perform practical validation of the ideas a prototype called *Repleo* is implemented based on the presented template grammars and evaluation strategy. The Repleo evaluator is generic with respect to the object language of a template grammar. It can be used for every template grammar constructed following the method of Chapter 4.

The implementation is separated in two components: the SGLR parser including the template grammars defined in SDF and the template evaluator. The template evaluator is written in Java and composed around a tree traversal, i.e. the *seval* function. The traversal has a connection with the SGLR parser to parse strings from the input data in case a substitution placeholder is detected. The evaluation process is stopped when an error occurs to prevent generation of syntactical incorrect code. The error handling uses the exception mechanism provided by Java. When an error is detected an error message is thrown, which contains the reason of the error and source code location in the template.

6.10 Case Studies

This section discusses examples of syntax safe templates. The first case study is the reimplementation of the PICO unparser of Section 4.4. The goal of this reimplementation is to show the benefits of implicit subtemplates.

In order to show that syntax safe templates are not limited to academic toy languages, we show three case studies for the industrially used object languages Java (general purpose programming language), XHTML (website markup language), and SQL (database query language). The choice of these object languages is based on their contemporary use in three-tier (web) applications; Java for the business layer, SQL for the database layer and XHTML for the presentation layer. Finally we discuss the definition of a template grammar and a template based on an object language supporting multiple languages. This example shows an object language based on Java with embedded SQL.

6.10.1 PICO Unparser

As mentioned the syntax safe evaluator provides implicit subtemplates and separator handling. This reduces the amount of syntax necessary to implement list generation and tree generation comparing to the textual based templates of Chapter 4. Figure 6.8 shows the reimplementation of the PICO unparser using the syntax safe evaluator. Most match-replace placeholders are nested, only the `expr` code is defined in a subtemplate, since it is called on three places in the template and replacing these calls with the code of the subtemplate `expr` will result in cloning. After refactoring the definition of the PICO unparser is 9 lines shorter than the text-based template PICO unparser implementation (see Section 4.4).

6.10.2 Java

The first example of a template with an industrially used object language is a Java template. This template generates data model classed in Java, i.e. Java classes with fields accompanied with getters and setters. The template is shown Figure 6.10. The template grammar used for parsing this template is shown in Figure 6.9.

The benefits of syntax safe templates are shown by a number of situations resulting in syntax errors in a text-based template environment in Table 6.1. This table summarizes errors that could be made in a template as shown in Figure 6.10 and in the input data as shown in Figure 6.11. There are two possible causes of syntax errors in a template based generation system. First, the object code of a template contains syntax errors and the generated code inherits these errors (errors A and B). Second, the data obtained from the input data to substitute a placeholder does not syntactically fit into the object code of the template (errors C, D and E). The first class of errors is usually detected during parsing the template, the second class of errors will be reported during the evaluation of the template. Unlike when using Repleo, the errors A and B are detected while parsing the template and the errors C, D and E are detected during the evaluation of the template.

```
template[
<: match :>
<: program( decls($decls), $stms ) =:>
  begin declare
  <: match $decls :>
  <: [] =:>
  <: [ decl( $id, $type ), $tail] =:>
  <: $id :> : <: match $type :>
                <: natural =:> natural
                <: string  =:> string
                <: nil-type =:> nil-type
              <: end :>,
  <: $tail sort:ID-TYPE* :>
  <: end :>
  ;
  <: match $stms :>
  <: [] =:>
  <: [ $head, $tail] =:>
    <: match $head :>
      <: assignment( $id, $expr ) =:>
          <: $id :> := <: expr( $expr ) :>
      <: while( $expr, $stms ) =:>
          while <: expr($expr) :> do
                  <: $stms sort:STATEMENT* :>
          od
      <: if( $expr, $thenstms, $elsestms ) =:>
          if <: expr($expr) :> then
                  <: $thenstms sort:STATEMENT* :>
          else
                  <: $elsestms sort:STATEMENT* :>
          fi
    <: end :>;
    <: $tail sort:STATEMENT* :>
  <: end :>
  end
<: end :>
]

expr[
<: match $expr :>
 <: natcon( $natcon ) =:> <: $natcon sort:EXP :>
 <: strcon( $strcon ) =:> <: $strcon sort:EXP :>
 <: id( $id ) =:> <: $id sort:EXP :>
 <: sub( $lhs, $rhs ) =:>
            <: $lhs sort:EXP :> - <: $rhs sort:EXP :>
 <: concat( $lhs, $rhs ) =:>
            <: $lhs sort:EXP :> || <: $rhs sort:EXP :>
 <: add( $lhs, $rhs ) =:>
            <: $lhs sort:EXP :> + <: $rhs sort:EXP :>
<: end :>
]
```

Figure 6.8 PICO abstract syntax tree unparser (syntax safe).

```
module Template-Java

imports Java
imports StartSymbol["CompilationUnit*" CompilationUnit*]
imports PlaceHolder["ID" ID]
imports PlaceHolder["Type" Type]
imports PlaceHolder["Modifier" Modifier]
imports PlaceHolder["ClassBodyDec*" ClassBodyDec*]
imports PlaceHolder["BlockStm*" BlockStm*]
```

Figure 6.9 Java template grammar.

```
1  class <: model2name1 :> {
2
3  <: model3cons1vis1 :> <: model2name1 :>(){}
4
5  <: match model4fields1 :>
6   <: [] =:>
7   <: [ field( $field ), $tail ] =:>
8    private <: $field2type1 :> <: $field1name1 :>;
9
10   <: $field2type1 :> <: "get" + $field1name1 :>(){
11     <: match $field1log3 :>
12       <: true =:> System.out.println("get" +
13                              <: "\"" + $field1name1 +
14                                  "\"" :>+"() is called.");
15     <: end :>
16     return <: $field1name1 :>;
17   }
18
19   void <: "set" + $field1name1 :>(
20                                <: $field2type1 :> value ){
21     <: $field1name1 :> = value;
22   }
23   <: $tail :>
24 <: end :>
25 }
```

Figure 6.10 A Java template.

	Substitute line	**by**	**creating error**
A	1	`clss <: model2name1 :> {`	class misspelled.
B	12	`1System.out.println("get"`	`1System` is not a valid identifier.
C	1	`class <: model1number1 :> {`	number not allowed as identifier.
D	Input, 3	`name("Shop-Client")`	identifier contains a dash.
E	Input, 4	`vis("abstract" )`	modifier abstract not allowed.

Table 6.1 Possible errors in template and input data.

```
model(
number(104),
name("Customer"),
cons(vis("public")),
fields([
  field(
    name("firstName"),
    type("String"),
    log(true)
  ),
  field(
    name("lastName"),
    type("String"),
    log(false)
  )
])
)
```

Figure 6.11 Input Data example for Java Template of Figure 6.10.

```
module Template-XHtml

imports XHtml
imports StartSymbol["XHtml" XHtml]
imports PlaceHolder["PCDATA" PCDATA]
imports PlaceHolder["CDATA" CDATA]
imports PlaceHolder["Quoted-CDATA" Quoted-CDATA]
imports PlaceHolder["XHtml-form-content-item*"
                    XHtml-form-content-item*]
```

Figure 6.12 XHTML template grammar.

6.10.3 XHTML

XHTML (Extensible Hypertext Markup Language) is a markup language based on XML and the Hypertext Markup Language (HTML), the language in which web pages are written. XHTML has a more strict syntax than HTML, and can be specified using a context-free syntax formalism. Templates are a common technique to render (X)HTML for web pages in web applications. The instantiation of a combination grammar for XHTML templates is shown in Figure 6.12. Based on the same input data of Figure 6.11, we provide a template to generate a basic XHTML form in Figure 6.13.

A screen shot of the output after evaluation of the XHTML template is shown in Figure 6.14. For every field in the input data an input field is generated in the web form.

The use of a template grammar for the template prevents syntax errors in the object code, such as a space between < and `input` at line 15 in Figure 6.13. The syntax safe evaluation results in the guarantee that the output is always a

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>
 <head>
  <title><: model2name1 + " form" :></title>
 </head>
 <body>
  <form action="/commit" method="post">
   <: match model4fields1 :>
    <: [] =:>
    <: [ field( $field ), $tail ] =:>
     <p>
      <: $field1name1 :> <br />
      <input type="text"
         name=<: "\"" + $field1name1 + "\"" :> size="20">
     </p>
     <: $tail :>
   <: end :>
    <p>
     <input type="submit" value="Submit">
     <input type="reset" value="Reset">
    </p>
  </form>
 </body>
</html>
```

Figure 6.13 XHTML Template.

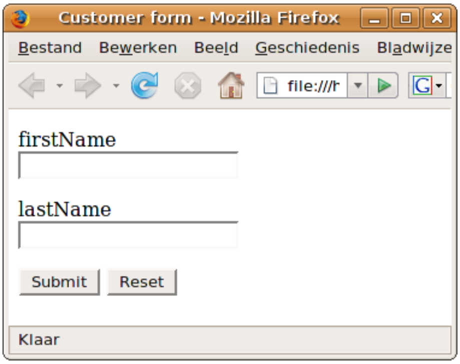


Figure 6.14 Output after evaluation XHTML template.

syntactically valid XHTML document. It can even be used to prevent injection attacks. This application of syntax safe evaluation is discussed in Section 7.7.

```
module Template-Sql

imports Sql
imports StartSymbol["SqlId" SqlId]
imports PlaceHolder["SqlId*" {SqlId ","}*]
```

Figure 6.15 SQL combination module.

```
SELECT
(
<: match model4fields1 :>
 <: [] =:>
 <: [ field( $field ), $tail ] =:>
  <: $field1name1 :> , <: $tail :>
<: end :>
)
FROM <: model2name1 :>
```

Figure 6.16 SQL Template.

6.10.4 SQL

SQL is a language in the domain of information systems for expressing database queries. The Listing 6.15 shows the combination module definition for parsing templates for SQL select statements.

The next example shows an SQL select statement template in Figure 6.16. This template can also be evaluated using the input data of Figure 6.11. The result of the evaluation of the SQL template is

```
SELECT  (firstName, lastName) FROM Customer
```

6.10.5 Multi-language Templates

We have shown a number of examples of instantiating a template grammar for different industrially used object languages. Sometimes program fragments contain multiple programming languages, like a general purpose language with an embedded domain specific language. The language containing foreign language artifacts is called the *host language* and the embedded language is called *guest language*. Usually these guest language fragments are embedded in strings, and considered as data in the host language. A compiler of the host language is usually not equipped to check the correctness of the embedded guest language code. The aim of this section is to show that syntax safe templates, where the object language is based on multiple sub languages, generates correct sentences with respect to all the the sub languages of the

```
class CustomerDao {
  public Collection getCustomers(){
    Collection items = new java.util.ArrayList();
    SQL q = <| SELECT (firstName,lastName)
                FROM Customer |>;
    ResultSet rs = con.query(q.toString());
    try {
      for (; rs.next(); ) {
        Customer item = new Customer();
        item.setfirstName(
          rs.getString("firstName"));
        item.setlastName(
          rs.getString("lastName"));
        items.add(item);
      }
    } catch (SQLException e) {
      e.printStackTrace();
      System.exit(1);
    }
    return items;
  }
}
```

Figure 6.17 Java with SQL code inside quotations.

object language. First we discuss StringBorg, an approach for guaranteeing that multiple languages in a program are syntactical correct.

StringBorg [25] is an approach solving the limitation of a compiler not supporting multiple languages. The system is able to check a host language with embedded guest language code. Quotations (<| ... |>) are used to switch from the host language, to the guest language(s), and anti-quotations (&{ ... }) to go back to the host language. StringBorg prevents that guest language artifacts from being syntactically incorrect and the approach especially aims at providing protection against injection attacks. A common example is a general purpose language, like Java, containing database queries expressed in SQL. Figure 6.17 shows a Java listing with embedded SQL using StringBorg to ensure syntactical correctness of both languages.

The consequence of using code with quotations is that the code cannot be compiled and/or executed without a tool interpreting these quotations. It is desirable to keep the build environment with as less as possible tools, since all these tools must be maintained. We present an approach introducing static syntax safety[1] for guest language fragments without adding new constructs to the syntax of the original host language. This can be achieved by a defining a grammar configured to detect guest languages without the quotations. We present an approach where the quotations are defined as function signatures in the host language. This approach is based on the assumption that most

[1] Static means that runtime injection of code into strings is not verified, we only check the correctness of the static defined code. In case of the JDBC this can be solved using prepared SQL statements, which remove the possibility of SQL injection.

```
 1 module Java-Sql
 2 imports Sql
 3 imports Java
 4 exports
 5   sorts SqlMethod SqlExpr MethodName
 6   context-free syntax
 7     SqlMethod "(" SqlExpr ")"     -> Expr {prefer}
 8     SqlMethodName                 -> SqlMethod
 9     SqlMethodName                 -> ID {reject}
10     AmbName "." SqlMethodName     -> SqlMethod
11     "prepareStatement"            -> SqlMethodName
12     "\"" Query "\""               -> SqlExpr
```

Figure 6.18 Combining Java and SQL.

guest language sentences are specified in an API function call. For example, most Java environments use JDBC [109] to establish a connection to a database. The guest language syntax is embedded in the host language grammar by adding a specialized function signature for the generic function signature in the grammar. This approach is less flexible than the StringBorg with respect to the places where guest language fragments can be defined. However, it adds no new language constructs to the host language, so no extra tooling is necessary to compile and execute the code. Furthermore, our approach does not prevent against dynamic injection attacks when running the generated code, but only finds errors in the static defined code. Still it is useful to guarantee that all code generated by a template is syntactically correct.

The grammar of Figure 6.18 shows the embedding of SQL in Java. It inserts a special grammar rule for the function `prepareStatement`. It is necessary that the function `prepareStatement` cannot be parsed as a normal function call. A reject rule is specified at line 9 to specify `prepareStatement` as a preferred keyword [23]. The `Java-Sql` grammar is able to parse Java with embedded SQL code in the `prepareStatement` function without adding new language constructions. The connection between the host language and the guest language should be based on API calls or other natural transition points. Some effort is needed to define a grammar module for this connection and to maintain it when the API changes. However API's such as JDBC do not change often or maintain backward compatibility [7].

Syntax safety for all language fragments is also crucial for templates. Syntax errors in guest language fragments of a template can easily remain undetected until runtime. The compiler of the host language does also not check the guest language fragments. From a grammar perspective the `Java-Sql` can be seen as a new language, in other words the union of languages results in a new language. This new language can be extended with placeholders in the same way as the original languages. The syntax safe template evaluator will guarantee that its output is a sentence of this new language, i.e. all the sub languages of the output are syntactical correct. Figure 6.19 shows the template

```
module Template-Java-Sql

imports Java-Sql
imports StartSymbol["CompilationUnit*" CompilationUnit*]
imports PlaceHolder["ID" ID]
imports PlaceHolder["Type" Type]
imports PlaceHolder["Modifier" Modifier]
imports PlaceHolder["ClassBodyDec*" ClassBodyDec*]
imports PlaceHolder["BlockStm*" BlockStm*]

imports PlaceHolder["SqlId" SqlId]
imports PlaceHolder["SqlId*" {SqlId ","}*]
```

Figure 6.19 Java-SQL template grammar.

```
class <: model2name1 + "Dao" :> {
 public Collection <: "get" + model2name1 + "s" :>(){
   Collection items = new java.util.ArrayList();
   ResultSet rs = con.prepareStatement(
    "SELECT (<: match model4fields1 :>
              <: [] =:>
              <: [ field( $field ), $tail ] =:>
                <: $field1name1 :> , <: $tail :>
              <: end :>) FROM <: model2name1 :>");
   try {
     for (; rs.next(); ) {
       <: model2name1 :> item = new  <: model2name1 :>();
       <: match model4fields1 :>
        <: [] =:>
        <: [ field( $field ), $tail ] =:>
         item.<: "set" + $field1name1 :>(
          rs.getString(<:"\""+$field1name1+"\"":>));
         <: $tail :>
        <: end :>
     items.add(item);
     }
   } catch (SQLException e) { ... }
   return items;
 }
}
```

Figure 6.20 Java-SQL template.

grammar for Java-Sql templates.

Figure 6.20 shows a Java-Sql template. This template can also be evaluated using the input data of the Listing 6.11. The result after evaluating the template is listed in Figure 6.21.

The generated code can be compiled without a special preprocessor and both host language code and guest language code are guaranteed to be syntactically correct. The template evaluator detects syntax errors in both languages, since the grammar contains production rules for both. For example, an error is generated when an identifier fits in a Java placeholder but not in an SQL

```
class CustomerDao {
 public Collection getCustomers(){
   Collection items = new java.util.ArrayList();
   ResultSet rs = con.prepareStatement(
        "SELECT (firstName, lastName)
         FROM Customer");
   try {
     for (; rs.next(); ) {
       Customer item = new  Customer();
       item.setfirstName(
         rs.getString("firstName"));
       item.setlastName(
         rs.getString("lastName"));
       items.add(item);
     }
   } catch (SQLException e) { ... }
   return items;
 }
}
```

Figure 6.21 Result of evaluation of Java-SQL template.

placeholder. This property forces us to use the greatest common divisor of the character classes of the Java identifiers and SQL identifiers in the input data. The advantage of using an object language grammar containing production rules for embedded languages is that sentences of these sub languages constructed during code generation are syntax safe. It is not an obstacle to use an object language based on multiple unified context-free languages, since syntax safe template evaluation only requires that the object language is context-free.

6.11 Related Work

The related work is separated in two sections. First, the related work on the topic of syntax safe templates is discussed. Second, other approaches of embedded languages are discussed.

6.11.1 Syntax Safe Templates

Two recent approaches for syntax safe templates are presented by Heidenreich et al. [57] and Wachsmuth [122]. The approach of Heidenreich et al. [57] is based on abstract syntax of templates. Heidenreich et al. designed a syntax safe template approach based on meta models. Meta models define the abstract syntax grammar of the template language and models are instantiations of these templates, comparable to an abstract syntax tree of a template. These meta models even allows one to go beyond syntax safety and perform some static semantic checking. However, a couple of problems are ignored when considering syntax safe template evaluation. First, Heidenreich et al. do not discuss a clear approach to handle ambiguities and leave this problem to the template grammar developer every time a new object language model is extended with metalanguage artifacts. Furthermore, the template evaluator uses models, i.e. a form of abstract syntax trees, and as a result no whitespace and other layout information is available. This can be a problem when layout sensitive code must be generated. Finally an unparser is necessary to transform a generated object language model into text. This unparser is automatically generated, but it is not by default ensured that the output is syntax safe.

Another approach is presented by Wachsmuth [122]. He discusses an approach to extend a target language grammar with metalanguage artifacts in order to obtain a template grammar, which guarantees syntax safe generation of code. It is similar to the constructing a template grammar discussed in Theorem 5.2.3. The approach of Wachsmuth statically enforces syntax correctness, since substitution placeholders are not supported and thus all output code is already defined in the template. Hence the relation with Theorem 5.2.3. This approach does not allow having substitution placeholders replacing themselves by string values from the input data. Substitution placeholders are not necessary for unparser-completeness (see Theorem 4.3.7). However, a metalanguage without substitution placeholders is less flexible in use, since the behavior of the substitution placeholder must be captured in a combination of match-replace like placeholders and subtemplates (see Section 4.3.5).

6.11.2 Embedded Languages

Embedding one computer language in another computer language is not a new phenomenon. Most times the embedded languages are considered as data, most times typed as strings. Several approaches go beyond handling these embedded languages as strings. StringBorg [25] is a system to to embed guest languages in a host language to provide a solution against injection attacks for arbitrary languages. StringBorg is the successor of MetaBorg [26]. It adds quotations to the grammar to indicate a switch between the host language and the guest language. The quotations are also the limitation of StringBorg; Source code must first be processed by StringBorg before it can be processed by the original compiler. In our approach of embedding languages the quotations are omitted. This allows us to compile the source code using the original compiler.

MetaBorg is a system to embed a guest language in a host language. It translates the guest language fragments into host language fragments. MetaBorg does not use explicit hedges to indicate the transitions between host language and guest language. The requirements for MetaBorg differ from our needs. We are not interested in expanding the embedded languages, but only in the syntactical correctness of the embedded code. Furthermore, we want to check the embedded language in the context where it is used in the host language. This is achieved by using function calls as quotations.

Another approach to embed a language is domain specific embedded compilers [79]. This is a technology to express a domain specific language, such as HASKELL/DB, in a high order typed language like Haskell. HASKELL/DB is a proprietary language to express database queries in Haskell, which are translated to SQL. The HASKELL/DB fragments are checked for syntax correctness and type correctness. The type safety is obtained by introducing *phantom types* for the guest language. Phantom typing is a technique to create annotations containing type information for the nonterminals in the parse tree of the HASKELL/DB code. This allows the Haskell type system to check the type correctness of the embedded language. The use of a proprietary language for SQL makes this concept less easy to use and maintain than a system based on the concrete syntax of SQL.

6.12 Conclusions

In this chapter we have presented syntax safe template evaluation. Syntax safe templates provide a mechanism to detect syntax errors during the generation of the code, instead of dealing with syntax errors at compile time. It prevents that placeholders in a template are replaced by syntactical incorrect sentences. The output code of a syntax safe template evaluator is a sentence of the intended output language.

The evaluation strategy is not dependent on the object language and does not need to be changed when another object language is used. It is even possible to use object code containing multiple languages.

We have implemented the presented approach in a prototype called *Repleo*. This prototype is used to validate the presented approach in a few case studies. In Chapter 7 we present a number of realistic case studies using Repleo.

7
Case Studies

Beside the theoretical discussion of templates, empirical validation is necessary to show that the ideas indeed work in practice. This chapter discusses the validation of the discussed syntax safe template evaluator by means of four case studies. These case studies show different aspects of the use of templates. A typical three tier web application is generated from a data model. Two case studies show a reimplementation of an existing code generator. The reimplemented code generators use a separated model transformation, resulting in less code and better maintainable code. Finally the use of syntax safe templates is shown for dynamic web page generation. Syntax safe templates provide a solution against cross-site scripting attacks.

7.1 Introduction

In the previous chapters we introduced (syntax safe) templates. This chapter discusses an empirical validation of the presented syntax safe template evaluator. The implementation of the syntax safe template evaluator Repleo is used for these case studies.

The case studies were chosen to show that our metalanguage in combination with the two-stage architecture, discussed in Section 7.2, results in better maintainable code and to show the benefits of syntax safety. We divided the case studies in four topics, covering different domains where code generation is used. Metrics, presented in Section 7.3, are used to provide indication of the size and complexity of case studies. We will use the results to compare the different case studies in the conclusions. The first case study discusses the generation of web applications back-ends in Section 7.4. The generated code is based on a three tier MVC architecture, the code generator must instantiate code for the different layers expressed in different languages from a single input model. The second case study, see Section 7.5, is the reimplementation of ApiGen [90]. ApiGen is an application to generate a Java API for creating, manipulating and querying tree-like data structures represented as ATerms. It covers the generation of Java code based on the *Factory* pattern and *Composite* pattern [45]. The third case study, see Section 7.6, is the reimplementation of NunniFSMGen[1]. NunniFSMGen is a tool to generate finite state machines from a transition table. It covers the generation of behavioral code based on the *state* design pattern [45] for different output languages. These three case studies are used to validate that the use of a two-stage architecture results in better separation of concerns. The final case study, presented in Section 7.7, shows that syntax safety can improve the safety of dynamic code generation in web applications. It covers code generation during the usage of an application, where syntax safety is used to reduce the possibility of security bugs.

We conclude this chapter with an overall conclusion. First, we start with the discussion of different implementation architectures of code generators used by the case studies.

7.2 Code Generator Architectures

We discuss three architectures used for implementing code generators. These architectures are used by both the original code generators and the reimplemented code generators. The first is the single-stage architecture. We continue with the two-stage architecture. It is based on two translation steps, where the input data is first translated to an intermediate representation before the final output code is generated. The two-stage architecture is used by all our implementations of template based code generators. The last discussed architecture is the model-view-controller architecture. This is a frequently applied architecture, also used for (web) applications generating code, where the template evaluator is used in the view component. The first case study generates code based on the model-view-controller architecture and the last case study uses templates to implement the view component. We discuss the architectures with their advantages and disadvantages.

7.2.1 Single-Stage Generator

The single-stage generator architecture is a basic implementation of a code generator. This architecture is easy to implement when starting with a new code generator. All processing and calculations are performed in a single module, without the use of an intermediate representation. The code generator directly emits code when parsing the input model.

The single-stage architecture has the drawback that the code is less maintainable and less reusable. Intermediate results and functions are not available

[1] http://sourceforge.net/projects/nunnifsmgen/ (accessed on November 30, 2010)

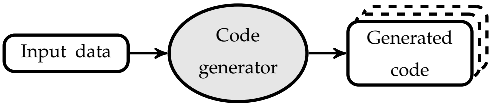


Figure 7.1 Single-stage architecture.

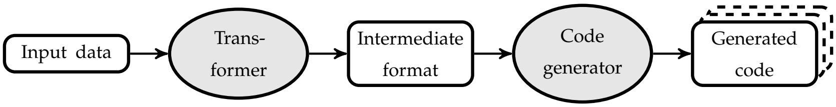


Figure 7.2 Two-stage architecture.

for reuse. The same calculations are performed every time they are necessary. Caching results of calculations will break the single-stage architecture, as the cached result is a kind of intermediate representation. Another side effect of this architecture is that code cloning can easily occur. At the moment the same calculations are necessary at different moments, it will result in a code clone. The single-stage architecture does not allow combining these code clones in a model transformation stage. Debugging is also more difficult, since everything is done in one phase and code cannot be tested in isolation. Figure 7.1 shows a visual representation of the single-stage architecture.

7.2.2 Two-Stage Generator

The two-stage architecture is based on two translation steps, where the input data is first translated to an intermediate representation before the final output code is generated. This architecture separates the code generation process in two (or possibly more) stages, namely a so called model transformation stage and the code emitting stage. The intermediate representation is used by the code emitter stage. In our case studies this code emitter stage is implemented as a set of templates in combination with a template evaluator. The model transformation stage is considered as one mapping, but depending on the necessary refining, it can be implemented using a number of sub mappings. Figure 7.2 shows the corresponding two-stage architecture.

In a two-stage generator, the translation of input model to output code is distributed over two components, namely a model transformation and code emitter(s). It is plausible that the input data, for example a list, tree, graph or concrete text, is most times compact without redundant information and that it has a higher level of abstraction than the generated output code. The greater the distance between the level of abstraction of the input data and the level of abstraction of the output code, the more calculations are needed to process the input data in order to construct the output code. In case multiple code emitters are used for generating the output code, a single-stage generator is implemented as a set of separated code generators using the same input model. It is likely, that this set of code generators share code clones, since the same distance between the levels of abstraction needs to be bridged. Compared to the single-stage architecture, the use of a two-stage architecture allows one to reduce the number of code clones in the generator code by combining frequently used calculations in the model transformation stage. The model transformation calculates an intermediate representation of the input data, which is used by the code emitters. The code cloning is reduced by specifying the non output language specific calculations in the model transformation.

It should be the aim to design the intermediate representation at a level of abstraction that it is not output language specific. When the intermediate representation does not contain output language specific information, it is possible to use it for multiple output languages. On the other hand, the intermediate representation should be close enough to the level of abstraction of the output code. At that point of abstraction, the code emitters act as a render component with minimal calculations, while the intermediate representation is still not output language specific. When the level of abstraction of the intermediate representation is too high, it is necessary to perform model transformation calculations in the code emitters, possibly resulting in clones between multiple code emitters. Although the intermediate representation should not contain output language specific artifacts, the intermediate representation can already be paradigm specific or has specific requirements of concepts, which an output language should support. For example, the intermediate representation implements a design pattern which can only be implemented using an object-oriented programming language.

The use of a two-stage approach is already a common architecture for implementing compilers [4, 5, 3]. For example, the GCC compiler[2] processes the input code and translates it to a representation belonging to the *register transfer language*. The register transfer language is the intermediate representation of GCC and is used to separate the compiler front-ends from the compiler back-ends. The intermediate representation is translated to assembler by the back-end of GCC. The register transfer language representations are still independent of the final target processor, so it can be used for different compiler back-ends for different processors.

7.2.3 Model-View-Controller Architecture

The *model-view-controller* architecture (MVC) [78] describes a decomposition of an application in three parts: *model*, *controller* and *view*, with their specific responsibility. It has a strong aim for separation of concerns, which improves

[2] http://gcc.gnu.org (accessed on November 30, 2010)

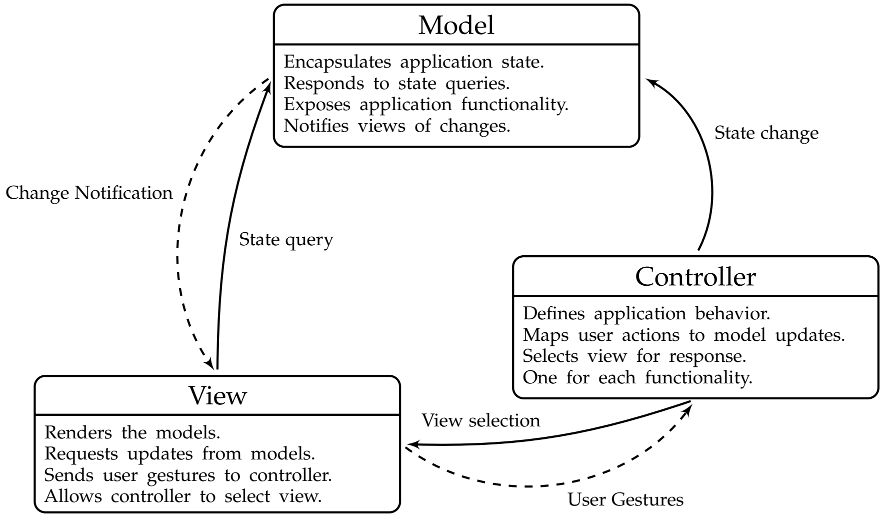

Model
Encapsulates application state.
Responds to state queries.
Exposes application functionality.
Notifies views of changes.
State change
Change Notification
State query
Controller
Defines application behavior.
Maps user actions to model updates.
Selects view for response.
One for each functionality.
View
Renders the models.
Requests updates from models.
Sends user gestures to controller.
Allows controller to select view.
View selection
User Gestures

Figure 7.3 Model-view-controller architecture.

re-usability and maintainability. The model is responsible for actually executing the calculations for the application domain and is distinct of the controller and view part of the application. The controller is used to send messages to the model, and provides the interface between the model with its associated views and the interactive user interface devices. The view deals with everything graphical on screen, printer, files and other devices, i.e. the output of the application. It uses data from the model component to render the output screen.

An MVC based application consists of a model and can have one or more views and controllers associated with it. The re-usability of the model is improved when it does not have knowledge of the views and controllers of the application, only the views and controllers need to have knowledge about their model explicitly. Figure 7.3 provides a visual sketch of the MVC architecture.

The original discussion of the MVC architecture [78] considers an end-user application, where the view is a (graphical) user interface and the controllers handle direct user input via keyboard or mouse. Beside the original context, the MVC architecture also fits for services generating code, like web applications. In MVC based web applications, the controller handles the requests and the views return generated HTML pages, PDF documents, etcetera. Often this view component of a web application is implemented as a template evaluator combined with templates.

The MVC architecture is used in the case studies of Section 7.4 and Section 7.7. The first case study discusses a code generator, where its output code is based on the MVC architecture. The last case study presents an MVC based web application using the syntax safe template evaluator for rendering the view.

7.3 Metrics

We present a couple of metrics in order to measure the volume of the code produced in the case studies and to quantify the readability of the code. The metrics are used to compare the size of all case studies. In the case studies of Section 7.5 and Section 7.6, they are also used to compare the original implementation and the reimplementation on a quantitative level.

The selected metrics are to indicate the volume of the code. The volume is measured using the lines of code metric. It gives a rough indication of the size of the code. The lines of code metric does not give accurate information about the amount of text in the source code, therefore the number of tokens is also counted.

Meta-programs have by nature a lot of non-alphanumeric characters to switch from object code to meta code and vice versa. These characters mainly influence the readability of the code. In order to quantify the readability of the code the number of non-alphanumeric characters used in the code is measured.

Tokens are counted instead of characters to measure the volume of the code. They abstract from the length of identifiers, as they are parsed as one token, otherwise the length of the identifiers could influence the comparison. The tokens are defined by four lexical classes, see the grammar of Figure 7.4 for the definition of them. The lexical classes are

◇ `ID` - detects identifiers and keywords in Java, C, SDF, ASF and shell scripts;

◇ `Bracket` - detects brackets, and initializes a token per bracket;

◇ `NonID` - detects non-alphanumeric character sequences, excluding brackets;

◇ `WS` - detects sequences of white space characters.

The grammar of Figure 7.4 is used to parse the source code of the code generator implementations. The result of the parser is a parse tree with a root node `Tokens` and for each token a child node. A function is defined iterating over this list of tokens, while counting the number of the different tokens. The number of non-alphanumeric characters is the sum of the number of `Bracket` tokens and `NonID` tokens. They are parsed as different nonterminals, since `Bracket` detects single characters and `NonID` detects the longest match for a character sequence.

```
module Tokens

exports
  sorts Text Bracket NonID ID WS
  context-free start-symbols Text
  context-free syntax
     (Bracket | NonID | ID | WS)*        -> Text

  lexical syntax
    [A-Za-z0-9\*\'\$]+                   -> ID
    [\"\{\}\[\]\<\>\(\)]                 -> Bracket
    ~[\"\{\}\[\]\<\>\(\)\ \t\n\rA-Za-z0-9\*\'\$]+
                                         -> NonID
    [\ \t\n\r]+                          -> WS

  context-free restrictions
    NonID -/- ~[\"\{\}\[\]\<\>\(\)\ \t\n\rA-Za-z0-9\*\'\$]
    WS -/- [\ \t\n\r]
    ID  -/- [A-Za-z0-9\*\'\$]
```

Figure 7.4 Grammar for token counter.

These metrics have a lot in common with the Halstead complexity metrics [50], based on the number of (distinct) operands and (distinct) operators. Unfortunately, we could not directly use these metrics in our case as we have a meta code and object code. It is not clear how to define operands and operators in a meta programming situation. Considering the object code fragments as operands is not sufficient. Our metrics are on a basic lexical level, where we do not consider the differences between metalanguage and object language.

7.4 A DSL for Web Information Systems

This section discusses an industrial case study using Repleo as template evaluator[3]. The topic of this case study is the generation of modules for a web-based information system. It has been carried out at MarketMind B.V.[4] in the context of a web application framework called WebManager. WebManager is a framework providing a plug-in based architecture for web application modules. These modules are model-view-controller based applications, which can be developed independently of each other.

The WebManager framework allows some reuse (on a modular level) and increases the ease of maintenance, but the company is still not satisfied about productivity. The problem of writing and maintaining WebManager modules is the amount of boilerplate code necessary to define them. When customers need

[3] This work is in detail discussed in the thesis of Smeets [105].

[4] http://www.marketmind.nl (accessed on November 30, 2010)

custom made modules to fit their needs it can be hard to offer a competitive price.

This case study investigates the possibility to use code generation to instantiate a WebManager module from a specification and to get rid of manually writing this boilerplate code. Since a WebManager module is based on a three tier MVC architecture, the code generator should instantiate code for the different layers expressed in different languages. This case study shows that syntax safe templates can be used to generate code for multiple output languages from a single input model. We give a brief overview of the approach.

7.4.1 Introduction

The domain specific language *WebManager Specification Language (WMSL)* is designed to specify web applications intended to be based on the *WebManager* framework of MarketMind. The code generator instantiates WebManager modules from a WMSL definition. A WebManager module is a 3-tier web application, containing a view, controller and a database component.

The WebManager Specification Language is a proprietary formalism specially designed for specifying WebManager modules. It is based on a textual representation of an entity–relationship model extended with a number of WebManager specific features. An example of such a WebManager feature is the support of localization of fields[5], like the text field of an `article` entity must be available in English and Dutch. An example WMSL specification is given in Figure 7.5. It declares an `Article` entity and `Author` entity with a number of fields. The primary key is the field marked with a hedge #. The relationship between the entities article and author is explicitly expressed by the relationship block.

A WMSL specification is translated to a database definition in SQL, a data access layer and data transfer layer, both implemented in C#. The data access layer contains the API to query the database. The data transfer layer provides the classes, based on the entities in the model, to hold the records of the database.

The implementation of the WMSL code generator is based on the two-stage architecture. First, the WMSL specification is refined using a couple of intermediate steps to obtain the input data for the templates. These transformations calculate the implementation of WebManager features and paradigm mismatches between the WMSL and the final target platform. An example of a paradigm mismatch is the support of many–to–many relations in WMSL, which cannot direct be specified in a database model. Figure 7.6 gives an overview of these transformations. The model transformations already fork the input data in branches for the different layers. This forking is necessary since

[5] Localization is the ability of translating a value into different natural languages for a specific country or region.

```
Module News
{
  Entity Article
  {
    ArticleId [#] : BigInt;
    Title         : String;
    Text          : String;
  }

  Entity Author
  {
    AuthorId [#]  : BigInt;
    AuthorName    : String;
  }

  Relationship ArticleAuthor
  {
    Article : n?;
    Author  : 1?;
  }
}
```

Figure 7.5 News article web module specification.

the layers are implemented using different platforms, not sharing the same expressiveness. For example, the data transfer model is finally implemented in C#, supporting many–to–many relations, while relational databases only support one–to–many relations. First the WMSL specification is normalized, like calculating canonical names for the entities. This normalized model is directly used for instantiating the input data for the data access layer. For the data access layer and database definition the normalized model is transformed to a so called logical data model. The main task of this transformation is to translate many–to–many relationships to two one–to–many relationships using an auxiliary table. Finally, this logical data model is used to instantiate the input data for the templates generating the database description and the data access layer.

For the generation of the three layers, i.e. the database, data access layer and data transfer layer, a set of templates is defined. The refined models, i.e. physical data model, data access model and data transfer model, are used as input data for these templates. From the physical data model SQL statements are generated. From the data access model the data access layer is generated, in this case C# code. From the data transfer model the data transfer objects are generated that are used in the front-end libraries and the web services layer. Figure 7.7 shows a snippet of the physical data model for the news article of Figure 7.5.

The input data for the templates are the abstract syntax trees resulting from parsing the refined models. The abstract syntax tree of the physical data model example is shown in Figure 7.8. It is obtained by parsing the listing of Figure 7.7. The structure of the abstract syntax tree is important for the

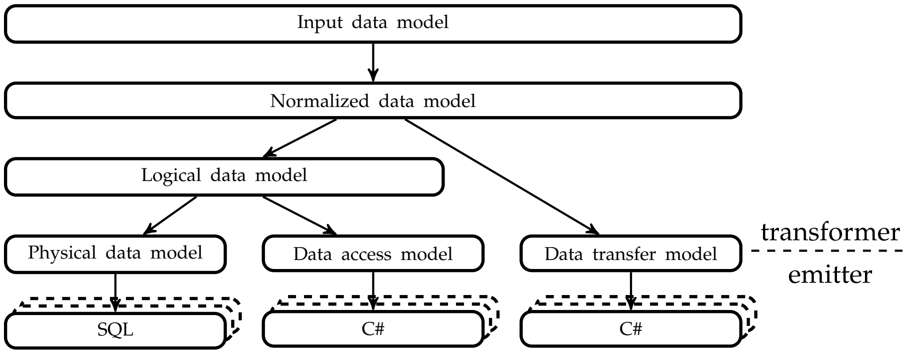


Figure 7.6 Code generation for web manager architecture.

```
Module News
{
 + Entity News . Article []
  {
   ArticleId [ # ] : BigInt;
   Title [] : String;
   Text [] : String;
   # News.Article.ArticleId;
  }
}
```

Figure 7.7 Physical data model for the news article of Figure 7.5.

template implementation as the meta code in the templates traverses it.

The code generation from WSML to a WebManager module is straightforward. The entities defined in the WSML are one-to-one implemented in the generated database definition and C# code, surrounded with the necessary boilerplate code. Complex recursive structures are not necessary to implement the WSML model; as a result the templates are trivial. This is not typical for this case study, but also EJBGen, a similar case study discussed by Herrington [58], contains quite trivial templates.

Even though the templates are straightforward, we highlight an example from the templates that are used to generate the SQL code for creating tables in a database. These templates use a physical data model as input, for example the model of Figure 7.8. The SQL template file is shown in Figure 7.9. The match-replace placeholder is used to match the `Module` node in the input data. The variable `$module` is assigned to the string "News", and the variable `$defs` is assigned to the list that is the second argument of the `Module` node. In this example this list contains exactly one element. The text shown in the template is written to the output file with the name `DataModel.$module.sql`, where

```
Module(
  ModuleName("News"),
  [
   CreateEntity(
     Entity(
       ModuleName("News"),
       EntityId("Article"),
       [],
       [
         Attribute(
           AttributeName("ArticleId"),
           AttributeOptions(
             [AttributeOptionPrimaryKey]),
           BigIntType),
         Attribute(
           AttributeName("Title"),
           AttributeOptions([]),
           StringType),
         Attribute(
           AttributeName("Text"),
           AttributeOptions([]),
           StringType),
         PrimaryKeyConstraint(
           ModuleName("News"),
           EntityId("Article"),
           AttributeName("ArticleId")
         )
       ]
     )
   )
  ]
)
```

Figure 7.8 AST of the physical data model from Figure 7.7.

`$module` is replaced with its value “News". The subtemplate call to the `stm` template passes the data of the variable `$defs` into the subtemplate.

The subtemplate `stmcreateentity`, see Figure 7.10, is called by the subtemplate `stm`. The `Entity` node is passed to this subtemplate and used in the match-replace placeholder to be matched against. Considering Figure 7.10, the variables `$m` and `$i` are matched against the input term and inserted into the `CREATE TABLE` statement. The `tabledefs` subtemplate generates the column names for the SQL statement from the attributes in the entity.

7.4.2 Evaluation

This case study showed that the proposed two-stage architecture and syntax safe templates are a fruitful combination to define a code generator for generating a three-tier web application. It is used for generating a couple of prototype applications, including a news site and a simple content management system. With respect to the current prototype implementation of the WMSL generator there are a number of things yet to be done. First and foremost Market Mind

```
<: match :>
<: Module(ModuleName($module), $defs) =:>
template[
DAL/<: $module :>/<: "DataModel." + $module + ".sql" :>,
BEGIN TRANSACTION;
GO
...
<: stm($defs) sort:Stm* :>
COMMIT TRANSACTION;
GO
]
<: end :>
```

Figure 7.9 Template for SQL code generation.

```
stmcreateentity[
<: match :>
<: Entity(ModuleName($m), EntityId($i), $opts, $attribs) =:>
CREATE TABLE [<: $m :>].[<: $i :>]
(
<: tabledefs($attribs) sort:TableDef* :>
)
<: end :>
]
```

Figure 7.10 Subtemplate that generates the create table statement.

would like to see that the WebManager modules are fully generated, including the view component.

Table 7.1 shows the metrics of the WMSL code generator implementation. We measured all 32 files of the original WMSL implementation, including the templates, and the grammars and rewrite rules belonging to the model transformation.

Metric	WMSL
Lines of Code	2,956
Lines of Code (without blank lines)	2,435
Tokens	34,638
Alphanumeric tokens	9,792
Non-alphanumeric tokens	14,483
White space tokens	10,363
Average number of tokens per line	14,23
Average number of non-alphanumeric tokens per line	5,95

Table 7.1 Metrics of WMSL code generator.

The metalanguage of the templates was not experienced as a limitation for the applicability of the templates for implementing the code generators for the different layers of a web application. It enforced the code generator developer to separate calculations from the specification of the view. Furthermore, having syntax safe templates enables the use of syntax highlighting in the template editor and detecting of misspelling while writing templates.

7.5 ApiGen

This case study covers the reimplementation of the Java back-end of ApiGen [90]. ApiGen [90] is a tool to generate a Java or C API for creating, manipulating and querying tree-like data structures represented as ATerms [22]. This case study uses the two-stage architecture and syntax safe templates in the setting of a complex code generator, since the Java back-end of ApiGen generates non-trivial Java code based on the *Factory* pattern and *Composite* pattern [45]. Furthermore having an old implementation and reimplementation allows us to compare them. We expect that the reimplemented code generator provides better separation of concerns and is more compact than the original implementation. The better separation of concerns is demonstrated by means of a code example. Metrics are used to show that the reimplemented code generator is indeed more compact.

7.5.1 Introduction

ApiGen finds its origin in the *ASF+SDF Meta-Environment* [21], an interactive development environment for program analysis and transformations. The ASF+SDF Meta-Environment provides various language processing components, such as a parser and a term rewrite engine. Data between these components is exchanged via ATerms, where these components expect that ATerms belong to a certain regular tree grammar. This regular tree grammar was not explicitly defined, but dictated by manually specified functions in an application programming interfaces (API) accepting the ATerm. Synchronizing these manually defined API's for the various components is a complex maintenance issue [90].

De Jong et al. [90] discuss an approach to remove this maintenance problem by generating an API for ATerms from a regular tree grammar or concrete syntax definition (SDF). ApiGen translates a regular tree grammar, specified in a format called *annotated data-type* or in short *ADT*, to an API for manipulating, reading and creating ATerms belonging to the language of the regular tree grammar. Generating an API from the ADT removes the need of error-prone handcrafted ATerms. The generated API also provides more safety when manipulating trees, since the node objects are not typed as a generic node, but

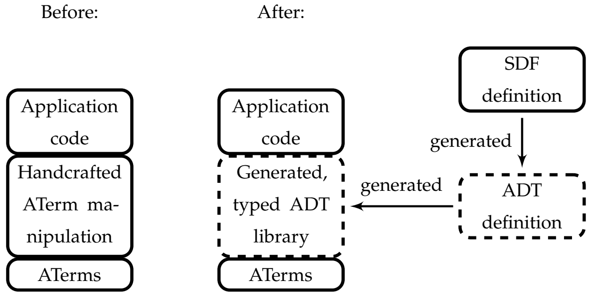


Figure 7.11 Overview of an application before and after introduction of a generated API.(From [90])

typed as its alternative, which is a subclass of its producing nonterminal. A schematic overview is shown in Figure 7.11. This figure also shows that an ADT can be derived from an SDF specification. The translation of SDF to ADT is out of the scope of this case study and we use the original tool `sdf2-to-adt` in order to generate an ADT from an SDF grammar.

7.5.2 Annotated Data Type

The input for ApiGen is the so called annotated data type, in short ADT. The ADT format is a formalism to define a set of legal ATerms, just like schema formalisms such as Document Type Definition and XML Schema for XML documents [14]. It can be considered as a regular tree grammar formalism, allowing trees with *infinite arity* [86]. Trees with infinite arity may contain symbols used with different arities, while the symbols in the previously discussed trees have a fixed arity. The ADT format finds it origin as an intermediate representation between an SDF definition and the generated API for manipulating parse trees belonging to that SDF definition. Therefore only the necessary information to generate an API is stored in the ADT, i.e. only the production rules and not the nonterminal declarations. Since the intended use of the ADT format is to represent the structure of parse trees belonging to an SDF definition, three kinds of production rules are supported:

◇ Production rules: `constructor( n, c, ATerm)`, where `n` is the nonterminal, `c` the alternative and `ATerm` the corresponding pattern. The couple of `n` and `c` must be unique in an ADT definition;

◇ Lists: `list(A, A')`, where `A` is the nonterminal and `A'` the element type;

◇ Separated lists: `separated-list( A, A', [ATerm+])`, where `A` is the nonterminal and `A'` the element type and `ATerm+` is a list of separators.

```
[
  list(PhoneBook, Entry),
  constructor( Entry, Home,
            home(<person(Name)>,<phone(PhoneNumber)>) ),
  constructor( Entry, Work,
            work(<company(Name)>,<phone(PhoneNumber)>) ),
  constructor( Name, Name, name(<string(str)>) ),
  constructor( PhoneNumber, Voice, voice(<integer(int)>)),
  constructor( PhoneNumber, Fax, fax(<integer(int)>))
]
```

Figure 7.12 The ADT definition for the phone book.

The running example for this case study is a phone book data-structure. It contains a list of entries, where the type of an entry can be person or company. Both person and company have the fields name and phone number. The phone number can be either a voice number or a fax number. Figure 7.12 shows the ADT definition for this phone book data-structure.

An instance of a phone book ATerm is presented below.

```
[
  home(
   name("Arnoldus"),
   voice(0205951616)
  ),
  work(
   name("Hogeschool"),
   fax(0205951620)
  )
]
```

It should be noted that the ADT formalism does not support the explicit definition of start symbols. Instead of defining a start symbol, ApiGen generates for all nonterminals a function to parse trees using that nonterminal as start symbol.

7.5.3 From ADT to an API

Before discussing the code generator implementation, we first discuss the mapping of an ADT to a Java API implementation. The code generated by ApiGen contains two different components:

- ◇ A data structure based on the regular tree grammar.
- ◇ A *factory* to create and manipulate trees stored in that data structure.

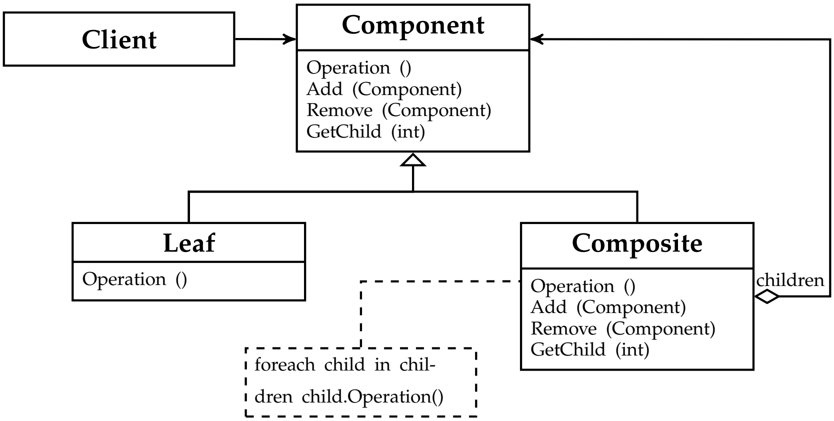


Figure 7.13 Composite design pattern [45].

These components of the generated API are based on the *composite* pattern and *factory* pattern as documented in [45].

A. Data Structure

The data structure implementation is based on the composite pattern, see Figure 7.13. It constructs a type structure to represent a tree in the form of objects connected to each other via a *has-a* relationship. The nonterminals in the regular tree grammar are implemented as abstract classes and the production rules are implemented as concrete subclasses of that nonterminal.

ApiGen generates the classes based on the composite pattern on top of the generic Java ATerm library. The connection between the ATerm Library and the concrete classes representing the nodes contains a number of inheritance steps. Two classes `AbstractType` and `AbstractListType` are generated to form the bridge between the generated API and the ATerm library. The first extends the class `ATermApplImpl` and the second the class `ATermListImpl`, which are members of the ATerm library representing an ATerm node and ATerm list node. The `AbstractType` and `AbstractListType` contain the default methods provided by every subclass of the generated API. The next layer generated by ApiGen is the abstract classes representing the nonterminals of the ADT. For each nonterminal an abstract class with the name of the nonterminal is generated. This generated abstract class extends `AbstractType` or in case the nonterminal represents a list, the abstract class extends `AbstractListType`. The nonterminal class contains the definition and default behavior of the accessor methods for that nonterminal. Figure 7.14 shows the important part of the generated class for the nonterminal `PhoneNumber`.

```
abstract public class PhoneNumber extends AbstractType {

  public PhoneNumber( ... ) {
     super( ... );
  }

  public boolean isEqual(PhoneNumber peer) {
    return super.isEqual(peer);
  }

  public boolean isSortPhoneNumber()  {
    return true;
  }

  public boolean isVoice() { return false;  }

  public boolean isFax() { return false; }

  public boolean hasInteger() { return false;  }

  public int getInteger() {
     throw new UnsupportedOperationException(
             "This PhoneNumber has no integer");
  }

  public PhoneNumber setInteger( int integer) {
     throw new IllegalArgumentException(
             "Illegal argument: integer");
  }

}
```

Figure 7.14 Snippet of the PhoneNumber class.

The final layer generated by ApiGen is the concrete classes for the different alternatives of a nonterminal. For each alternative a class with the name of the alternative is generated. This class extends the abstract class of the nonterminal belonging to the alternative, where all the accessor methods are implemented. These classes are finally used to instantiate trees belonging to the tree language defined by the ADT. Figure 7.15 shows the concrete class for the alternative `Fax` of the nonterminal `PhoneNumber` generated by ApiGen. In case of the alternative for `Voice` the same listing is generated, where `Fax` is replaced by `Voice`.

B. Factory

The second generated component is based on the factory pattern, see Figure 7.16, to instantiate trees based on the generated data structure. This generated factory provides methods to create, parse, manipulate and export trees conforming to the ADT. It is obligatory to use a factory for instantiating ATerm based trees in order to provide *maximal subterm sharing*; only one instance of any subterm exists in memory. The Java ATerm library provides this factory

```
public class Fax extends PhoneNumber {

  public Fax( ... ) { super( ... ); }
  private static int index_integer = 0;

  public shared.SharedObject duplicate() { ... }

  public boolean equivalent(shared.SharedObject peer)
    { ... }

  protected aterm.ATermAppl make(
      aterm.AFun fun, aterm.ATerm[] args,
      aterm.ATermList annos) {
    return
      getPhonebookFactory()
         .makePhoneNumber_Fax(fun, args, annos);
  }

  public aterm.ATerm toTerm() {
    if (term == null) {
      term = getPhonebookFactory().toTerm(this);
    }
    return term;
  }

  public boolean isFax() { return true; }

  public boolean hasInteger() { return true; }

  public phonebook.types.PhoneNumber setInteger(int  integer) {
    return (phonebook.types.PhoneNumber) super.setArgument(
      getFactory().makeInt(integer), index_integer);
  }

  public int getInteger() {
    return ((aterm.ATermInt)
               getArgument( index_integer )).getInt();
  }

  public aterm.ATermAppl setArgument(aterm.ATerm arg, int i)
    { ... }

  protected int hashFunction() { ... }

}
```

Figure 7.15 Snippet of the Fax alternative for the PhoneNumber nonterminal.

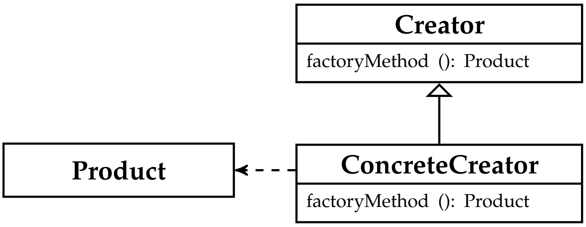


Figure 7.16 Factory design pattern [45].

by the implementation called "`ATermFactory`". In case we generate an API for an ATerm based tree language, a factory must be created supporting the instantiation of trees for that tree language. Therefore, ApiGen generates a layer on top of the `ATermFactory` containing `make` methods to instantiate (sub)trees using the classes of the generated data structure.

The generated factory provides special `make` methods for the nodes defined in the ADT. These `make` methods form the abstraction of the ATerm library, as it is not necessary to know the underlying ATerm representation to create trees defined by the ADT. When called, these `make` methods instantiate nodes based on the classes of the generated data structure and encapsulate the administration to achieve maximal subterm sharing. In case of list types, the factory class also provides the list operations `reverse`, `concat` and `append`. Beside the `make` methods, the factory class also contains methods to instantiate a tree from a string, to serialize it from a tree to a string, and cast methods to transform a tree to a generic ATerm representation and vice versa. Figure 7.17 shows a snippet of the generated factory for the phone book example.

```
public phonebook.types.phonenumber.Voice
    makePhoneNumber_Voice(int _integer ) {
  aterm.ATerm[] args = new aterm.ATerm[] {
      factory.makeInt( _integer )
  };
  return makePhoneNumber_Voice(
      fun_PhoneNumber_Voice, args, factory.getEmpty());
}

public phonebook.types.phonenumber.Voice
    makePhoneNumber_Voice(int _integer,
    aterm.ATermList annos) {
  aterm.ATerm[] args = new aterm.ATerm[] {
      factory.makeInt( _integer )
  };
  return makePhoneNumber_Voice(
           fun_PhoneNumber_Voice, args, annos);
}
```

Figure 7.17 Snippet of the generated factory.

7.5.4 Original Code Generator

The original Java implementation of ApiGen is based on a two-stage architecture. The first stage reads the ADT file and instantiates an in-memory instantiation of it. The second stage consists of a number of code emitter classes using the in-memory model. Based on reverse engineering the original implementation of ApiGen, both stages are discussed in more detail.

The first stage is the ADT reader. It reads an ADT specification, containing the specification of the tree structure, from file. The ADT is an ATerm and as a result the in-memory representation of an ADT is implemented by a generated API. The ADT reader calls the factory to instantiate the in-memory representation of an ADT file. Besides reading the ADT file, the first stage also executes a couple of model transformations. For example, the constructor rules in the ADT contain an ATerm pattern belonging to that node. The model transformation extracts the ATerm placeholders to collect the fields for that constructor rule. Figure 7.18 shows the field collection method of the model transformation. It is a recursive function traversing an ATerm pattern and instantiating a field object for every ATerm placeholder occurring in the pattern. For example the fields `company` and `phone` with their respective arguments `Name` and `PhoneNumber` are extracted from the ATerm pattern of the `work` alternative

```
work(<company(Name)>,<phone(PhoneNumber)>).
```

The second stage of ApiGen is responsible for generating the output code. For every kind of classes ApiGen generates a code emitter is written. ApiGen contains seven code emitter classes: `FactoryGenerator`, `AbstractListGenerator`, `AbstractTypeGenerator`, `ListTypeGenerator`, `SeparatedListTypeGenerator`, `TypeGenerator` and `AlternativeGenerator`. The emitter classes are implemented as `println` generators, which results in a mix of Java used as object code and Java used as meta code. Figure 7.19 shows the `genMakeMethod` method of the `FactoryGenerator` class. This method is responsible for generating the `make` methods of the factory class, as shown in Figure 7.17.

For each alternative declared by the constructor rules in the ADT a set of `make` methods are generated. Considering the code snippet of Figure 7.19, the following actions are executed. The first statements, in lines 3-9, construct the identifiers used in the generated code by calling a number of helper functions. At line 11-12, the output code is instantiated by printing a string to the output buffer, internally redirect to the output file. The `if`-statement is used to select the generation of a forwarded `make` method in case the factory is defined by an imported ADT module. Helper functions such as `buildActualTypedAltArgumentList` are used to reduce code clones in the generator self. At line 31 an `if`-statement is used to ensure the separator token is only generated when the alternative has fields.

```
private void extractFields(ATerm t) {
  AFun fun;
  ATermAppl appl;
  ATermList list;
  switch (t.getType()) {
    case ATerm.APPL :
      appl = (ATermAppl) t;
      fun = appl.getAFun();
      for (int i = 0; i < fun.getArity(); i++) {
        extractFields(appl.getArgument(i));
      }

      break;
    case ATerm.LIST :
      list = (ATermList) t;
      for (int i = 0; !list.isEmpty(); i++) {
        extractFields(list.getFirst());
        list = list.getNext();
      }
      break;
    case ATerm.PLACEHOLDER :
      ATerm ph = ((ATermPlaceholder) t).getPlaceholder();
      if (ph.getType() == ATerm.LIST) {
        ...
        addField(fieldId, fieldType);
      } else if (ph.getType() == ATerm.APPL) {
        ...
        addField(fieldId, fieldType);
      } else {
        throw new
          RuntimeException("illegal field spec: " + t);
      }
      break;
    default :
      break;
  }
}
```

Figure 7.18 Extraction of fields from ATerm pattern.

The original ApiGen implementation is based on a two-stage architecture. However, the distribution of tasks for the model transformation and code emitter in the original implementation of ApiGen is not optimal. Some model transformations are executed during code generation. For example Figure 7.20 shows a model transformation task inside the `AlternativeGenerator` class at line 4. This method `genAltFieldIndexMembers` is called for every type defined in the ADT. The type object contains all fields possible for that type. When this method is called, the statement at line 4 fetches all fields for the current alternative of the type. In our opinion this filtering should be done at model transformation phase, for example the class `Alternative` should provide a list of its associated fields. Although this filtering seems innocent, it is a call back to the model, so a wrong implementation of the `altFieldIterator` could change the model. `altFieldIterator` is a member of the type `Type` and has direct access to the private fields of `Type`. It is possible to change the values of these private fields, resulting in a change of the model during code generation.

```
private void genMakeMethod(Type type,
 Alternative alt, boolean forwarding, String moduleName) {
  JavaGenerationParameters params =
               getJavaGenerationParameters();
  String altClassName =
               AlternativeGenerator.qualifiedClassName(
                  params, type, alt);
  String makeMethodName = "make" + concatTypeAlt(type, alt);
  String funVar = funVariable(type, alt);

  print("   public " + altClassName
              + ' ' + makeMethodName + "(");
  printFormalTypedAltArgumentList(type, alt);
  println(") {");
  if (!forwarding) {
    print(
     "     aterm.ATerm[] args = new aterm.ATerm[] {");
    printActualTypedArgumentList(type, alt);
    println("};");
    println("     return " + makeMethodName +
      "(" + funVar + ", args, factory.getEmpty());");
  } else {
    ...
  }
  println("   }");
  println();

  print("   public " + altClassName
                         + ' ' + makeMethodName + "(");
  printFormalTypedAltArgumentList(type, alt);
  if(type.altFieldIterator(alt.getId()).hasNext())
                                              print(", ");
  println("aterm.ATermList annos) {");
  if (!forwarding) {
    ...
  } else {
    ...
  }
  println("   }");
  println();
}
```

Figure 7.19 Small part of the ApiGen code emitter.

Furthermore, `genAltFieldIndexMembers` cannot be expressed in Repleo, as a compare operator and a mechanism to store values is necessary.

7.5.5 Reimplemented Code Generator

We reimplemented ApiGen using the two-stage architecture with a strict separation between the model transformation stage and code emitter stage. The model transformation are defined as a set of rewrite rules using a term rewriting system (ASF+SDF) and the code emitter is implemented using syntax safe templates. The model transformations used in the presented case studies are mainly tree transformations. Term rewriting provides a powerful computa-

```
private void genAltFieldIndexMembers(Type type,
                                     Alternative alt) {
  Iterator<Field> fields =
              type.altFieldIterator(alt.getId());
  int argnr = 0;
  while (fields.hasNext()) {
   Field field = fields.next();
   String fieldId = getFieldIndex(field.getId());
   println("  private static int " + fieldId
         + " = " + argnr + ";");
   argnr++;
  }
}
```

Figure 7.20 Example of model transformation in code emitter.

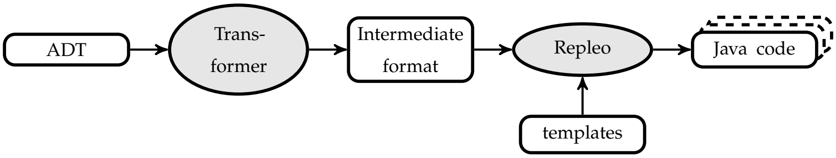


Figure 7.21 Generation scheme: from ADT to code.

tional paradigm to express these tree transformation. Figure 7.21 shows the architecture of the reimplementation. The rectangular shapes denote (intermediate) files and the circular shapes denote the transformation engines. The model transformation reads the ADT input model and computes the intermediate representation, also an ATerm. The second stage is defined using a set of templates accepting the intermediate representation as input data. The reimplementation of ApiGen uses a small bash script to connect the model transformation stage to the code emitters.

The intermediate representation of the reimplementation already implements the design patterns used in the generated code. We will show it using the intermediate representation of the phone book example, see Figure 7.22. The intermediate representation has a list of types, divided into `type` and `list` kinds. These types represent the nonterminals and are implemented as abstract classes. Considering Figure 7.14, an abstract class with the name of the type `PhoneNumber` is generated, containing `is` methods for every alternative of the type and a default get and set method for the fields. The alternatives of a type are used for generating the concrete classes, as shown in Figure 7.15. One can notice cloning of fields between the level of the type node and alternative nodes. The metalanguage of our templates is not capable to calculate the fields for an alternative; as a result the field nodes are stored multiple times in the input data. Since this intermediate representation is also used for the generation of the factory class, the alternatives contain an ATerm pattern corresponding with

```
[
type(
 "Entry",
 [
  alternative( "Home" , 2, "home(<term>,<term>)",
   [
    field( "person", notreserved("Name"), 0 ),
    field( "phone", notreserved("PhoneNumber"), 1 )
   ]),
  alternative( "Work" , 2, "work(<term>,<term>)",
   [
    field( "company", notreserved("Name"), 0 ),
    field( "phone", notreserved("PhoneNumber"), 1 )
   ])
 ],
 [
  field( "person", notreserved("Name") ),
  field( "phone", notreserved("PhoneNumber") ),
  field( "company", notreserved("Name") )
 ]
),
type(
 "Name",
 [
  alternative( "Name" , 1, "name(<str>)",
   [
    field( "string", reserved(str), 0 )
   ])
 ],
 [
    field( "string", reserved(str) )
 ]),
type(
 "PhoneNumber",
 [
  alternative( "Voice" , 1, "voice(<int>)",
   [
    field( "integer", reserved(int), 0 )
   ]),
  alternative( "Fax" , 1, "fax(<int>)",
   [
    field( "integer", reserved(int), 0 )
   ])
 ],
 [
  field( "integer", reserved(int) )
 ]
),
list( "PhoneBook", notreserved("Entry") )
]
```

Figure 7.22 The intermediate representation of the phone book.

the node.

The model transformation is responsible for translating the ADT input model to the intermediate representation. Considering the ADT of Figure 7.12 and the intermediate representation of Figure 7.22, the model transformation has collected all constructor rules for a type in a single node, where the different constructor rules are represented as alternatives for that type. Besides collecting

```
getFieldsForAlt(< $IdCon($IdCon') >, < $FieldAlt*, $NatCon >) =
<
$FieldAlt*,
field( unparse-to-string($IdCon),
       getTypeImplementation($IdCon'),
       $NatCon ),
$NatCon + 1
>
```

Figure 7.23 Extraction of fields from ATerm pattern.

the types, the model transformation extracts the fields from the ATerm pattern specified in each constructor rule. Additionally the ATerm pattern is translated to the pattern used in the factory class, i.e. the placeholders are replaced with generic ATerm placeholders.

Figure 7.23 shows the extraction of fields for an ATerm pattern belonging to an alternative for a type. It is the reimplementation of the code of Figure 7.18 using ASF+SDF. It is specified as a traversal walking along the ATerm pattern, where this rule declares a match on an ATerm placeholder. The identifiers having a dollar-sign prefix are declared as variables. The traversal stops when it detects an ATerm placeholder, specified by the pattern < `$IdCon($IdCon')` >. At a match it returns an instantiated field for the intermediate representation. Note, the `$NatCon` is a helper argument in order to provide an index number to the field.

The second stage is responsible for the actual code generation and is implemented using syntax safe templates. Seven templates, equal to the seven emitter classes in the original implementation and having the same tasks, are defined. The templates are constructed by extracting the object code from the original class and add placeholders when necessary. Figure 7.24 shows a snippet of the template generating the factory class. This snippet is responsible for generating the `make` methods, see Figure 7.17 for the result when using the input data of Figure 7.22. These methods are generated for each alternative of each type. The loop over the alternatives is expressed by two match-replace placeholders. This example also uses the built-in metalanguage functions `_lc` and `_cc`. These are shorthand notations for transforming a string to lowercase or camel-case, i.e. the first letter is capitalized. From a Puritan point of view, these functions should not be available in the metalanguage. The different layouts of identifiers should be stored in the input data. In practice, this would lead to a lot of variants of lexical layouts of identifiers in the input data, while most string manipulations are necessary to comply to the layout convention of the object language[6]. The availability of these functions makes it possible to express the

[6] It is possible to express the semantics of the string manipulation functions in a couple of subtemplates in combination with match-replace placeholders. The implementation would be a variant of the template given in Section 4.3.5.

```
...
template[
...
<: match :>
<: [ type($type,$alternatives,$fields), $types ] =:>
 <: match $alternatives :>
 <: [ alternative($altname,$arity,$pattern, $altfields),
                                    $alternatives ] =:>
 public <:$apiname:>.types.<:_lc($type):>.<:_cc($altname):>
   <: "make" + $type + "_" + _cc($altname) :>
                          ( <: genArguments( $altfields ) :> ) {
   aterm.ATerm[] args =
          new aterm.ATerm[] { <: genArray( $altfields ) :>  };
   return <: "make" + $type + "_" + _cc($altname) :>(
          <: "fun_" + $type + "_" + _cc($altname) sort:Expr:>,
                                     args, factory.getEmpty());
  }

 public <:$apiname:>.types.<:_lc($type):>.<:_cc($altname):>
            <: "make" + $type + "_" + _cc($altname) :>
               (<: genArguments( $altfields ) :>,
                                      aterm.ATermList annos) {
  aterm.ATerm[] args =
              new aterm.ATerm[] { <: genArray( $altfields ) :> };
  return <: "make" + $type + "_" + _cc($altname) :>
    ( <: "fun_" + $type + "_" + _cc($altname) sort:Expr:>,
                                                   args, annos);
 }
 ...
 <: $alternatives sort:ClassBodyDec*:>
 <: [ ] =:>
 <: end :>
<: $types sort:ClassBodyDec*:>
...
<: [] =:>
<: end :>
...
]
...
```

Figure 7.24 Small part of the ApiGen code emitter.

layout requirements in the template, instead of having them in the input data.

7.5.6 Comparing the Old and the New Implementation

We compare the original implementation and reimplementation on architectural level and code level. On architectural level, both implementations are based on the two-stage architecture. The original implementation does not have a concrete syntax for the intermediate representation, but uses a class structure for the objects storing the derived information. However, the original implementation does not have a strict separation between the two-stages as the code emitters call the model transformation during generation of the code. In the reimplemented ApiGen calling the model transformation from the code emitter is impossible as there is only a one-way link between the model transformation and the templates. For example, our restricted metalanguages forces us the specify the calculation performed in the code emitter method of Figure 7.20 in the model transformation stage.

On code level the original ApiGen implementation is Java based, while the reimplementation is implemented using ASF+SDF for the model transformation and syntax safe templates for the code emitters. The model transformation is a function reading the ADT, which is a tree, with as output the intermediate representation, which is also a tree. Tree rewriting can be expressed compactly using a term rewriting system, especially since the used term rewriting system ASF+SDF has native support for traversal functions [19]. The model transformation phase in the reimplementation is specified with a total of 21 functions based on 54 (sub) equations. The original model transformation implementation using Java is more verbose and uses a total of 3112 lines of code (without blank lines) spread over 34 files. An example of the effect of reduction of lines of code is the refactoring of the snippet of Figure 7.18 to the equation in the model transformation of the reimplementation shown in Figure 7.23.

In the original implementation the code emitter stage is implemented using `println()` statements. The use of templates in the reimplementation results in a smaller implementation in volume. Common tasks like file handling and separator handling are covered by the template evaluator. The advantage of having templates is that the object code is the main language in the document, instead of the meta code in case of the original implementation.

The Table 7.2 shows the result of measuring the old implementation of ApiGen and the reimplementation. We measured all 57 files of the original ApiGen implementation[7]. The total number of files of the reimplementation is 14, including grammar definitions, model transformations, shell script and templates.

Considering Table 7.2 we see that the number of lines of code of the reimplementation is almost a third of the original implementation and the number of tokens is reduced 2.5 times. The reduction of the volume of the code is a result of three differences between the original implementation and the reimplementation. First the model transformation is expressed in a term rewriting formalism instead of Java. Second, the original implementation uses a generated API for the ADT format, while the reimplementation only contains a grammar definition for it. The last reason is that original implementation contains code for output file handling, which is encapsulated by the template evaluator in the reimplementation.

Beside the reduction of tokens, the ratio's between lines of code[8] and tokens is not improved. The number of tokens per line of code in the reimplementation

[7] The original implementation also provided a C code emitter. We removed these classes from the project.

[8] We used the lines of code without blank lines.

Metric	Original	Reimplementation
Lines of Code	8,789	2,975
Lines of Code (without blank lines)	7,296	2,361
Tokens	84,230	33,254
Alphanumeric tokens	25,986	7,572
Non-alphanumeric tokens	33,704	15,669
White space tokens	24,540	10,013
Average number of tokens per line	11.5	14.08
Average number of non-alphanumeric tokens per line	4.62	6.64

Table 7.2 Metrics of ApiGen and the reimplementation.

is even higher than the original implementation. Thereby the number of non-alphanumeric tokens per line is increased by 43%. This is caused by the fact that the original implementation of ApiGen has a limited number of functions containing object code. The programmers of ApiGen have aimed for the reduction of code clones on the level of the object code specification. Having less strings results in a lower level of non-alphanumeric tokens. At first sight when inspecting the code, we experienced that the original implementation was better readable than the reimplementation. However, at the moment we noticed that the reimplementation contains Java code in strings; it is finally harder to read the code it generates than the implementation using templates. This is amplified, since the object code in ApiGen is presented as small chunks and one must analyze the flow graph to understand how the output code is constructed [29].

7.5.7 Related Work

A similar approach is used in JastAdd [54] to translate regular tree grammars into a data structure implementation. JastAdd uses the *interpreter* pattern, see Figure 7.25. The difference between the composite pattern and interpreter pattern is mainly the intention, where the composite pattern is a structural pattern, while the interpreter pattern is a behavioral pattern. This is reflected by the fact that the composite pattern classes have accessor methods and construction methods, while the interpreter pattern has an evaluation method including an evaluation context.

7.5.8 Evaluation

We discussed the reimplementation of ApiGen using templates. It is possible to re-implement a complex code generator like ApiGen using syntax safe templates and the metalanguage as presented in Chapter 4. We showed that use of a two-stage architecture, without a bidirectional connection between both

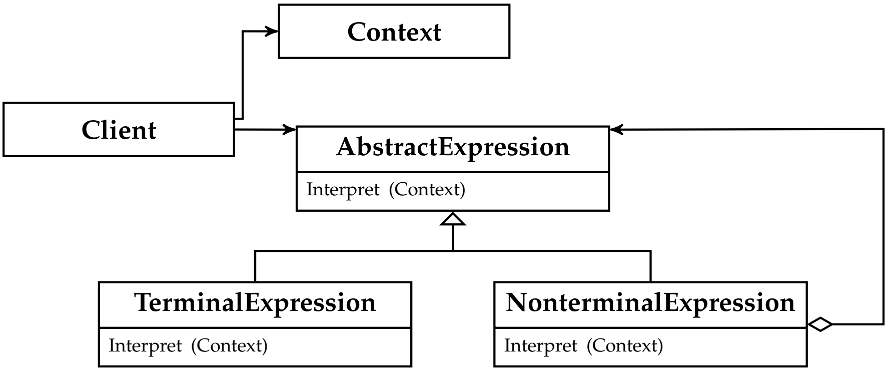


Figure 7.25 Interpreter design pattern [45].

stages, improved the separation of concerns between the model transformation and the code emitters. The use of formalisms better suitable for the intended task, i.e. term rewriting for the model transformation and templates for the code emitters, results in a more compact definition of the code generator and it results in better maintainability [106]. The number of lines of code of the reimplementation is almost a third of the original implementation and the number of tokens is reduced 2.5 times. Although the code is more compact, the functionality of ApiGen is not changed and syntax safety is even introduced.

7.6 NunniFSMGen

This section discusses the reimplementation of NunniFSMGen[9] using syntax safe templates. NunniFSMGen is a tool to translate a specification of a finite state machine into an implementation for Java, C or C++. It uses the state design pattern [45] to implement the state machine in the different output languages.

This case study covers the generation of code based on a behavioral design pattern. Furthermore it shows that the same intermediate representation can be used for different output languages, as long as they support the implementation of the state design pattern. This section shows the reimplementation of NunniFSMGen using a parser, model transformation and templates. We first give an overview of the original approach of NunniFSMGen. Next the reimplementation of NunniFSMGen is presented. At the end we discuss the main differences between the original implementation and the new template based implementation.

[9] http://sourceforge.net/projects/nunnifsmgen/ (accessed on November 30, 2010)

7.6.1 Finite State Machines

We start with a formal definition of finite state machines, before discussing the input format of NunniFSMGen. Hereafter we relate their input format to the formal concept of finite state machines [5].

Definition 7.6.1. (Deterministic finite state machine) A deterministic finite state machine is a 5-tuple $M = \langle \Sigma, Q, q0, \delta, F \rangle$, where:

Σ is the input alphabet,

Q is a finite, non-empty set of states,

$q0$ is an initial state and $q0 \in Q$,

δ is the state-transition function: $\delta : Q \times \Sigma \to Q$,

F is the set of final states, a (possibly empty) subset of Q.

Example 7.6.2. Let `M` be a finite state machine, where

$\Sigma = \{\text{activate}, \text{deactivate}, \text{hotenough}, \text{maintenance}\}$,
$Q = \{\text{STANDBY}, \text{WARMINGUP}, \text{ERROR}, \text{MAINTENANCE}\}$,
$q0 = \text{STANDBY}$, $F = \{\}$ and the transition function:

$$\begin{aligned}
\delta(\text{STANDBY}, \text{activate}) &= \text{WARMINGUP}\\
\delta(\text{STANDBY}, \text{hotenough}) &= \text{ERROR}\\
\delta(\text{STANDBY}, \text{maintenance}) &= \text{MAINTENANCE}\\
\delta(\text{STANDBY}, \text{deactivate}) &= \text{STANDBY}\\
\delta(\text{WARMINGUP}, \text{activate}) &= \text{WARMINGUP}\\
\delta(\text{WARMINGUP}, \text{deactivate}) &= \text{STANDBY}\\
\delta(\text{WARMINGUP}, \text{hotenough}) &= \text{STANDBY}\\
\delta(\text{WARMINGUP}, \text{maintenance}) &= \text{MAINTENANCE}\\
\delta(\text{ERROR}, \text{activate}) &= \text{ERROR}\\
\delta(\text{ERROR}, \text{deactivate}) &= \text{ERROR}\\
\delta(\text{ERROR}, \text{hotenough}) &= \text{ERROR}\\
\delta(\text{ERROR}, \text{maintenance}) &= \text{MAINTENANCE}\\
\delta(\text{MAINTENANCE}, \text{activate}) &= \text{STANDBY}\\
\delta(\text{MAINTENANCE}, \text{hotenough}) &= \text{MAINTENANCE}\\
\delta(\text{MAINTENANCE}, \text{maintenance}) &= \text{MAINTENANCE}\\
\delta(\text{MAINTENANCE}, \text{deactivate}) &= \text{MAINTENANCE}
\end{aligned}$$

This state machine describes the behavior of a central heating system (see Figure 7.26 for a graphical representation). It is an example state machine

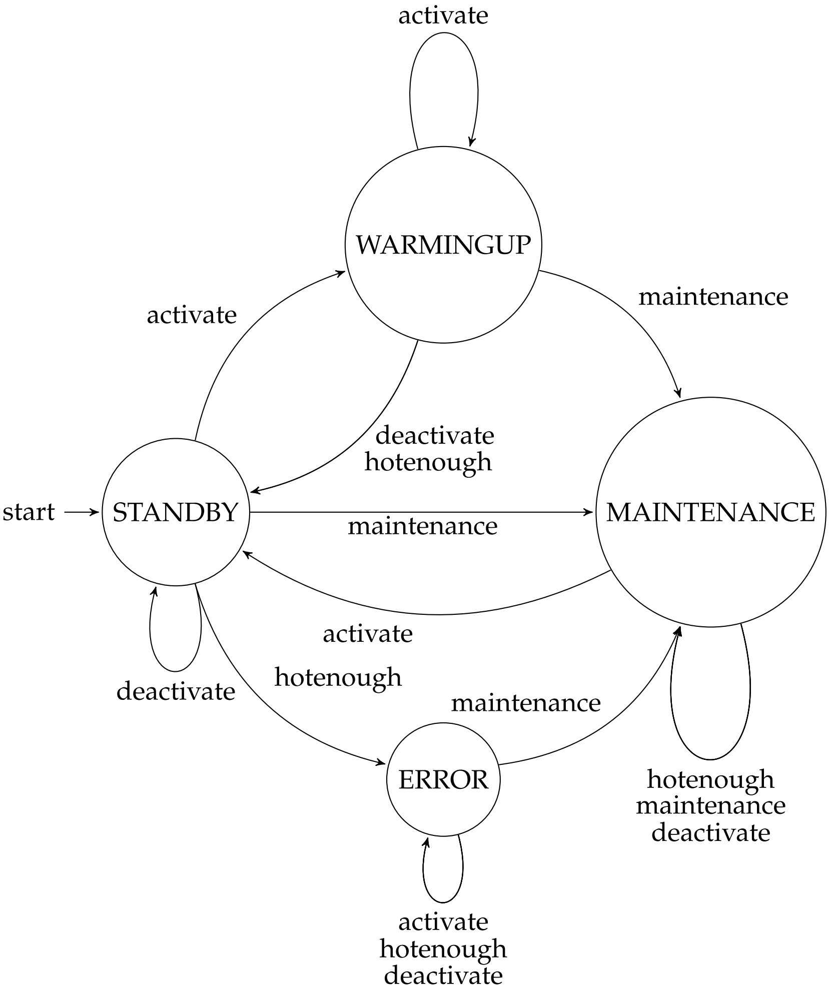


Figure 7.26 Graphical representation of the state machine.

distributed along with the source code of the original NunniFSMGen[10]implementation.

7.6.2 NunniFSMGen Input Model

The input model used by NunniFSMGen is not a 5-tuple of a state machine as defined in Paragraph 7.6.1, but a transition table. The transition table is a

[10] http://sourceforge.net/projects/nunnifsmgen/ (accessed on November 30, 2010)

set of transition rules of the form `startingState event nextState action`. The tuple of `startingState` and `event` is equal to the left-hand side of the transition rules of Definition 7.6.1. The name `event` is chosen instead of `token` to indicate it is an event driven state machine. The right-hand side of the transition rule corresponds to the `nextState`. An additional feature is to specify a method call hooked to a transition using the `action` field. This method call is invoked when the state machines uses that transition rule. Furthermore if the `nextState` is a dash `-`, then a transition rule will not cause a change of state. The dash `-` can also be used in the `action` field to specify that no action is required when the transition is executed. The action can also contain an exclamation mark `!`. The exclamation mark defines that the action must throw an exception and after that the state machine will go the *error state*. The states and alphabet are declared implicitly by means of the transition rules.

The transition table also supports the declaration of parameters to specify (optional) properties, such as the error state. The following properties are available:

- `Context`: Name of the FSM. All resulting classes are named after this name using it as prefix.
- `InitialState`: The initial state of the FSM.
- `ErrorState`: The error state of the FSM. This is a required field in case an error action '`!`' is specified.
- `Package`(only Java): Name of the package of the generated code.
- `EventParamType`(optional): parameter type passed on event methods.
- `Copyright`(optional): a file containing a copyright text to be included at the top of each generated file.

We did not implement the optional properties of the transition table for the reimplementation of NunniFSMGen.

We specified the grammar showed in Figure 7.27 for the original transition table format. The grammar also has constructor information to translate the parse tree to an abstract syntax tree. We omitted the definition of the lexical sorts except for `Id`. The definition of `Id` is based on the character class shared by all output languages of NunniFSMGen. When the character class of one output language is richer than the other output language(s), it is possible that code generated for the first language is well-formed, while the code generated for the next language is syntactically incorrect. For the output languages Java, C and C++ is it no issue, since they share the same character class for identifiers. Furthermore, we limited the set of allowed identifiers by defining *reject* rules for `Id` to prevent collisions with keywords of the output languages, such as the `if` at line 22. Since we use syntax safe templates, these reject rules are

```
module FSMGen

hiddens
  context-free start-symbols Rules
exports
  sorts Rules Rule NextState Action
  context-free syntax
    Rule*                        -> Rules {cons("rules")}
    "Context" Id                 -> Rule {cons("context")}
    "InitialState" Id            -> Rule {cons("initial")}
    "ErrorState" Id              -> Rule {cons("error")}
    "Package" PackageName        -> Rule {cons("package")}
    "Copyright" FileName         -> Rule {cons("copyright")}
    "EventParamType" Id          -> Rule {cons("paramtype")}
    Id Id NextState Action       -> Rule {cons("transition")}
    Id                           -> NextState {cons("nextstate")}
    "-"                          -> NextState {cons("nonextstate")}
    Id                           -> Action {cons("action")}
    "-"                          -> Action {cons("noaction")}
    "!"                          -> Action {cons("erroraction")}

    "if"                         -> Id {reject}

exports
  sorts Id
  lexical syntax
        [a-zA-Z\_][a-zA-Z\_0-9]* -> Id
  lexical restrictions
        Id -/- [a-zA-Z\_0-9]
```

Figure 7.27 Context-free grammar for NunniFSMGen transition table.

superfluous, but the advantage of having them already in the grammar of the input model is that errors are earlier detected in the generation process. Instead of using *reject* rules in the transition table grammar, we could add a prefix to the identifiers in the generated code. We did not use that approach to have a clear mapping between the transition table and the generated code. The original implementation of NunniFSMGen does not contain any checks for these errors and just considers the values in the input model as strings.

In accordance with the state machine definition of Definition 7.6.1, NunniFSMGen requires that a transition rule is defined for each pair existing in the Cartesian product of states and events, even when transition and action are empty. In other words, the number of transition rules $|transition\ rules|$ must be equal to $|states| * |events|$. A NunniFSMGen transition table for the central heating system of Example 7.6.2 is shown in Figure 7.28. The abstract syntax tree of this transition table is given in Figure 7.29. It is obtained by desugaring the parse result of the transition table. The model transformation presented in Section 7.6.3 uses this abstract syntax tree as input.

```
Context        Heater
InitialState   STANDBY
ErrorState     ERROR
Package        examples.heater

STANDBY        activate      WARMINGUP     warmup
STANDBY        deactivate    –             –
STANDBY        hotenough     ERROR         !
STANDBY        maintenance   MAINTENANCE   maintain
WARMINGUP      activate      –             –
WARMINGUP      deactivate    STANDBY       heateroff
WARMINGUP      hotenough     STANDBY       heateroff
WARMINGUP      maintenance   MAINTENANCE   heateroff
ERROR          activate      –             –
ERROR          deactivate    –             –
ERROR          hotenough     –             –
ERROR          maintenance   MAINTENANCE   –
MAINTENANCE    activate      STANDBY       initialize
MAINTENANCE    deactivate    –             –
MAINTENANCE    hotenough     –             –
MAINTENANCE    maintenance   –             –
```

Figure 7.28 Transition table for the central heating system of Example 7.6.2.

```
rules([
 context("Heater"),
 initial("STANDBY"),
 error("ERROR"),
 package("examples.heater"),
 transition("STANDBY","activate",
    nextstate("WARMINGUP"),action("warmup")),
 ...
 transition("MAINTENANCE","maintenance",
                      nonextstate,noaction)
])
```

Figure 7.29 Part of the abstract syntax tree of the transition table of Example 7.6.2.

7.6.3 State Machine Implementation

NunniFSMGen translates a transition table into an implementation based on the state design pattern [45]. The state design pattern is given in Figure 7.30 and the different classes correspond to the following functionality:

◇ `Context`

It defines the abstract methods to handle events and is the interface for components using the state machine.

It defines a private `changeState` method replacing the current state object by a new one.

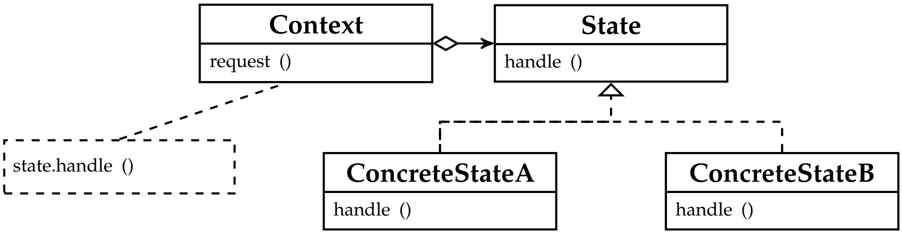


Figure 7.30 State design pattern.

◇ `State`

It defines an interface for all classes that represent different operational states.

◇ `ConcreteStates`

Each subclass implements a behavior associated with a state.

It handles the different events invoked via the `Context` object. The handle call has the `Context` object as argument and when a transition is defined, the `changeState` method is called with an object of the new concrete state. If an action is defined, this action is called before the `changeState` method is invoked.

The UML class diagram of the central heating system of the Java code generated by NunniFSMGen is shown in Figure 7.31. NunniFSMGen implements the state pattern using the transition table, where the events are the handles and the states are implemented as concrete states. It slightly differs from the original standard state pattern. First the context class `HeaterFSM` overrides a class `Heater`. The class `Heater` contains the implementation of the action methods and is used by the states to invoke the action methods at a state transition. The inheritance of the class `Heater` is used to allow editing of the action method bodies in the `Heater` class. The second difference with the original state design pattern is the object `o` in the argument of the event handlers. This argument is an optional object, which can be used by the action methods as context information.

The C++ and C implementation are almost similar, except that the C implementation uses a struct to store the state object. Furthermore, C has no native support for exception handling, which is simulated by a return value.

The transition table cannot be used directly for generating the code when looking to the UML diagram of Figure 7.31. The state pattern has a hierarchical structure, where each state implements handlers for each event, while the transition table is list of vectors pointing from the `startState` to the `nextState`.

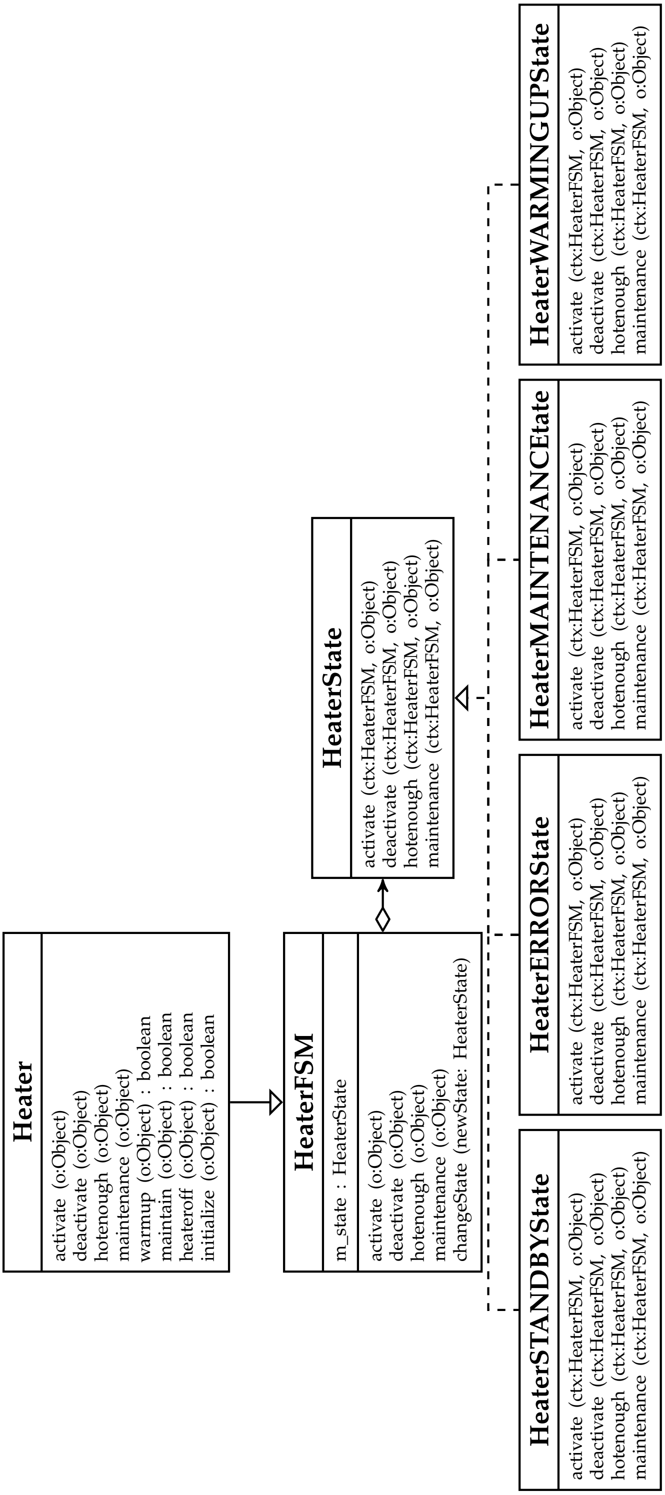

Heater
activate (o:Object)
deactivate (o:Object)
hotenough (o:Object)
maintenance (o:Object)
warmup (o:Object) : boolean
maintain (o:Object) : boolean
heateroff (o:Object) : boolean
initialize (o:Object) : boolean
HeaterFSM
m_state : HeaterState
activate (o:Object)
deactivate (o:Object)
hotenough (o:Object)
maintenance (o:Object)
changeState (newState: HeaterState)
HeaterState
activate (ctx:HeaterFSM, o:Object)
deactivate (ctx:HeaterFSM, o:Object)
hotenough (ctx:HeaterFSM, o:Object)
maintenance (ctx:HeaterFSM, o:Object)
HeaterSTANDBYState
activate (ctx:HeaterFSM, o:Object)
deactivate (ctx:HeaterFSM, o:Object)
hotenough (ctx:HeaterFSM, o:Object)
maintenance (ctx:HeaterFSM, o:Object)
HeaterERRORState
activate (ctx:HeaterFSM, o:Object)
deactivate (ctx:HeaterFSM, o:Object)
hotenough (ctx:HeaterFSM, o:Object)
maintenance (ctx:HeaterFSM, o:Object)
HeaterMAINTENANCEtate
activate (ctx:HeaterFSM, o:Object)
deactivate (ctx:HeaterFSM, o:Object)
hotenough (ctx:HeaterFSM, o:Object)
maintenance (ctx:HeaterFSM, o:Object)
HeaterWARMINGUPState
activate (ctx:HeaterFSM, o:Object)
deactivate (ctx:HeaterFSM, o:Object)
hotenough (ctx:HeaterFSM, o:Object)
maintenance (ctx:HeaterFSM, o:Object)

Figure 7.31 Classes generated from the heater transition table.

```
hiddens
  start-symbols AFSM

exports
  sorts AFSM Context InitialState ErrorState
        Package CopyRight ParamType Transitions
        Events Actions Transition Event
        NextState Action State

  syntax
    afsm( Context, InitialState,
          ErrorState, Package, CopyRight,
          ParamTypem, Transitions, Events,
          Actions )
                                   -> AFSM
    context( Id )                  -> Context
    initialstate( Id )             -> InitialState
    errorstate( Id )               -> ErrorState
    package( PackageName )         -> Package
    copyright( FileName )          -> CopyRight
    paramtype( Id )                -> ParamType
    transitions( Transition* )     -> Transitions
    transition( State, events( Event* )
                                   -> Transition
    event( Event, NextState, Action )
                                   -> Event
    events( Id* )                  -> Events
    actions( Id* )                 -> Actions
    Id                             -> State
    Id                             -> Event
    Id                             -> NextState
    Id                             -> Action
    StrCon                         -> Id
    StrCon                         -> FileName
    StrCon                         -> PackageName
```

Figure 7.32 Regular Tree Grammar of abstract implementation of the state design pattern.

A model transformation is necessary to map the vector based transition table to the hierarchical based state design pattern.

First an intermediate language of abstract implementation of the state design pattern is defined by the regular tree grammar of Figure 7.32. The important artifacts are the `initialstate`, the set of `events`, the set of `actions` and the list of `transitions`. The first three elements directly map on elements of the 5-tuple of the formal definition of a state machine. The element `transition` defines the transition for a state for every possible event. Instantiations of this abstract implementation can be translated to a real implementation as long as the state design pattern can be expressed in the output language.

We defined a model transformation to transform the abstract syntax tree of the transition table to the abstract implementation of the state design pattern. This transformation is based on the following, here informally defined, rules:

◇ Propagate all properties to the output model;

```
afsm(
 context("Heater" ),
 initial("STANDBY"),
 error("ERROR"),
 package("examples.heater"),
 copyright(""),
 paramtype(""),
 transitions([
  transition( "STANDBY", events([
    event( "activate",
               nextstate("WARMINGUP"), action("warmup") ),
    event( "deactivate", nonextstate, noaction ),
    event( "hotenough", nextstate("ERROR"), erroraction ),
    event( "maintenance", nextstate("MAINTENANCE"),
                                action("maintain") ) ])),
    ...
 ]),
 events([ "activate", "deactivate",
          "hotenough", "maintenance" ]),
 actions([ "warmup","maintain","heateroff", "initialize" ])
)
```

Figure 7.33 Abstract implementation of the state design pattern of the heater transition table of Example 7.6.2.

- ◇ Collect all unique events from the transition rules and store them in the set `events`;
- ◇ Collect all unique actions from the transition rules and store them in the set `actions`;
- ◇ Make for each unique state a new `transition` rule and add for each event a triple with the `event`, the `nextState` and the `action`. Collect all these rules in the set `transitions`.

This model transformation is straight-forward. This is reflected in an ASF based implementation of the model transformation containing seven equations based on 18 sub-equations. The abstract implementation of the state design pattern of the central heating system is shown in Figure 7.33.

7.6.4 Original Code Generator

The original implementation of NunniFSMGen is a printf based generator written in Java. Its main class contains a simple parser for the input file, constructing a one-to-one in memory representation of the transition table without any kind of rewriting or transformation. Based on the selected output language a code generator class is instantiated provided with the loaded input data. For each output language a code generator class is implemented containing all the generator logic and object code. We can consider such a code generator class as a single-stage code generator, since the different code generator classes share a lot of mutual shared code not factorized out in a model transformation. Furthermore, since it is a single-stage code generator, the model transformations are mostly entangled between object code artifacts. During the initialization of the code generator, only the set of events and the set of states are calculated. An example of the entanglement model transformation is the generation of the particular code for an event. This code is combined with the calculation of all events for a given state.

NunniFSMGen supports different configurations for an event:

- No transition, no action;
- No transition, with action;
- Transition to new state without action;
- Transition to new state with action;
- Transition to `errorState` with error action.

Considering the original implementation of NunniFSMGen, it selects the different implementations of the required behavior via a set of conditions. Since NunniFSMGen exists of three almost independent single-stage code emitters, the set of conditions are cloned between the different code emitters. The NunniFSMGen code emitters for C++ and Java are almost identical except for the object code. In case of the code emitter for C, the meta code is almost the same, but the way the object code implements the exception handling differs from the C++ and Java version.

7.6.5 Reimplemented Code Generator

The reimplementation of NunniFSMGen is based on a two-stage architecture using a parser, model transformation phase and templates. The input model parser and model transformation are output language independent, while the templates contain the output language specific code. For each output language, i.e. C, C++ and Java, a set of templates is defined. The mutual shared code in the templates is limited to the meta code responsible for traversing the input data tree. All templates use the same abstract representation of the state machine as input data. Tailored model transformations for a specific output language are unnecessary. As a result the model transformation is not longer entangled in the output language specific part of the code generator. The architecture of the reimplemented NunniFSMGen is shown in Figure 7.34.

Figures 7.35, 7.36 and 7.37 show snippets of the reimplementation using syntax safe templates. The use of syntax safe templates has some consequences over the use of a text based generator. For example, the C and C++ grammars have a couple of *plus list* nonterminals, which requires that at least one item

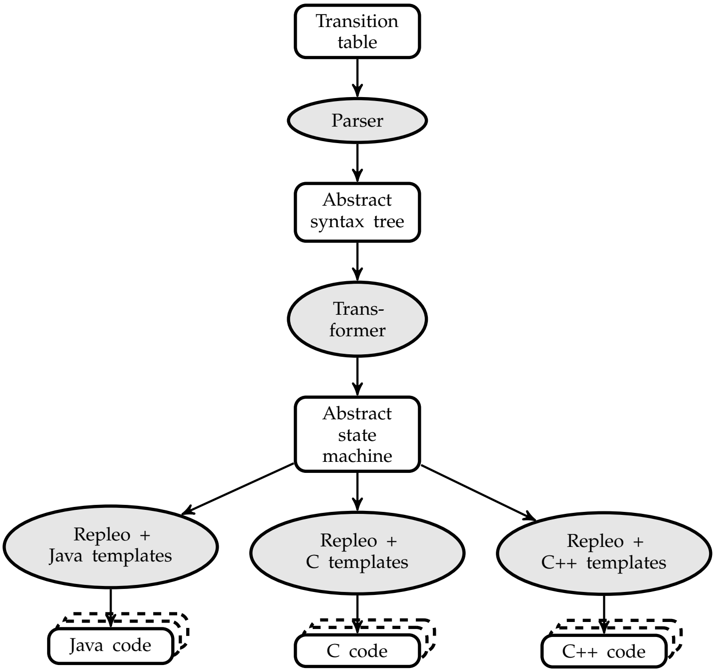


Figure 7.34 Architecture of the reimplemented NunniFSMGen.

must be inserted in that list. Placeholders parsed as a child of a plus list are typed as a plus list, resulting in the requirement that it should at least generate one element. The placeholders and template evaluator traversal are context-free, so they have no notion of the surrounding elements. The template evaluator cannot determine whether the list is empty or not. Therefore a plus list placeholder must return at least one element. An example is the semi-colon in the match-replace placeholder at line 46 of Figure 7.35.

Another example of a consequence of using a syntax safe approach, and thus also for syntax safe templates, is the cloning visible in lines 9-21 of Figure 7.37. The if-statement in the object language is defined twice, first with an else part and second without the else part. Syntax safe templates require that the if-statement is a complete grammar element. The else part is not defined as an optional nonterminal in the used C grammar, thus the code must be defined twice.

Furthermore, these snippets show the metavariable `$root` a number of times. It is used to obtain global information, like the FSM context name while processing a subtree of the input data. Finally, the Java template of Figure 7.36 shows how multiple files are generated for every concrete state by the match-replace placeholder surrounding the template `template`.

```
template[
...
<: match $events1events1 :>
  <: [event( $event, $nextstate, $action )] =:>
<: eventcode() :>
  <: [event( $event, $nextstate, $action ),$eventlist] =:>
<: eventcode() :>
<: $eventlist :>
<: end :>
..
]

eventcode[
void <: $root1afsm1context1 + $state + "State" :>::<: $event :>
    ( <: $root1afsm1context1 + "FSM" :> *ctx,
                void *o ) throw (LogicError) {
  <: match $action :>
    <: erroraction =:>
      ctx->changeState(
         <: $root1afsm1context1 + $root1afsm1error5
            + "State" :>::instance() );
      throw LogicError();
    <: action($actionname) =:>
      try {
         ctx-><: $actionname :>( o );
      }
      catch( LogicError &e ) {
        ctx->changeState(
          <: $root1afsm1context1 + $root1afsm1error5
             + "State" :>::instance() );
      throw;
      }
      <: nextstatetmp($nextstate)  :>
    <: noaction =:>
      <: nextstatetmp($nextstate) :>
  <: end :>
}
]

nextstatetmp[
 <: match :>
   <: nextstate($nextstatename) =:>
     ctx->changeState(
        <: $root1afsm1context1 + $nextstatename
           + "State" :>::instance() );
   <: nonextstate =:> ;
 <: end :>
]
```

Figure 7.35 C++ version of the template implementation.

```
<: match afsm1transitions6 :>
  <: [ transition( $state, $events), $transitions ] =:>
template[
<: $root1afsm1context1 + $state + "State.java" :>,
class <: $root1afsm1context1 + $state + "State" :>
          extends <: $root1afsm1context1 + "State" :>
{
...
}
]
<: $transitions :>
<: [] =:>
<: end :>
```

Figure 7.36 Java snippet of the template implementation.

```
eventcode[
static int <: $root1afsm1context1 + $state + "State" + $event :>
       ( struct <: $root1afsm1context1 + "FSM" :> *fsm,
         void * o ) {
  int ret = 0;
  <: match $action :>
    ...
    <: nextstate($nextstatename) =:>
      if ( ret < 0 )
          fsm->changeState( fsm,
             &<: "m_" + $root1afsm1context1
                 + $root1afsm1error5 + "State" :> );
      else
          fsm->changeState( fsm,
             &<: "m_" + $root1afsm1context1
                  + $nextstatename + "State" :> );
    <: nonextstate =:>
      if ( ret < 0 )
          fsm->changeState( fsm,
             &<: "m_" + $root1afsm1context1
                 + $root1afsm1error5 + "State" :> );
    ...
  <: end :>
  return ret;
}
]
```

Figure 7.37 C snippet of the template implementation.

7.6.6 Comparing the Old and the New Implementation

We compare both implementations on architectural level and code level. On architectural level, the new implementation of NunniFSMGen is based on a two-stage architecture, where the old implementation almost directly generates code from the transition table in a single-stage architecture. The reimplementation offers better separation of concerns, it separates the model transformation from the code generation. The first stage parses and rewrites the transition table in order to get an abstract implementation of the state design pattern. The second stage is responsible for generating the concrete code for the different output languages. We expect that the more strict separation of concerns reduces the work for adding a new output language than adding a new output language in the original implementation. In the original implementation also the model transformation has to be reimplemented. Furthermore, the original code generator contains code clones between the different code emitters for the different output language. These code clones are solved by introducing a model transformation stage for the mutual shared code and leaving the case specific code in the templates.

At code level, the original implementation shows a lot of entanglement of different code artifacts in a single file. The presented examples of the original implementations contain statements for the model transformation, statements for the code generation phase and strings containing the object code in a single file. The object code syntax has a lot in common with the meta language syntax, which make it hard to distinguish the different code artifacts. The cocktail of these different code fragments in a single file is confusing and is hard to read, test and maintain. When looking to the templates of the new implementation, it is still a complex part of the code generator. However, the template based implementation shows a better syntactical difference between the meta code and object code. Furthermore the object code is not encapsulated in single line strings, so the object code is not obfuscated by brackets and quotes, making it easier to read and review.

The Table 7.3 shows the metrics for the old implementation of NunniFSMGen and the reimplementation. We measured all the nine files of the original NunniFSMGen and only stripped the license block from the code. The set of files measured of the reimplementation are the grammar definitions, model transformations and templates. Total number of files of the reimplementation is 19.

Considering Table 7.3 we see that the number of lines of codes without blanks is almost halved, while the reimplemented ApiGen offers syntax safety. Considering the number of tokens of the reimplementation it is more than halved with respect to the original implementation. Remarkable is that the average number of non-alphanumeric characters per line is not altered. In comparison with the original implementation of ApiGen, the code emitter classes of NunniFSMGen show a lot of string statements containing object code. Since string

Metric	Original	Reimplementation
Lines of Code	1,602	1,005
Lines of Code (without blank lines)	1,430	738
Tokens	22,024	10,438
Alphanumeric tokens	5,820	2,393
Non-alphanumeric tokens	9,189	4,762
White space tokens	7,015	3,283
Average number of tokens per line	15.4	14.14
Average number of non-alphanumeric tokens per line	6.43	6.45

Table 7.3 Metrics of NunniFSMGen and the reimplementation.

statements contain a lot of brackets and quotes, the result is that the ratio of non-alphanumeric tokens per line is not lower in the original implementation.

7.6.7 Evaluation

We discussed the reimplementation of NunniFSMGen using templates. The original implementation is a single-stage generator for each output language. The reimplementation is based on a two-stage architecture, where the output language is selected via another set of templates in the second stage.

The use of the two-stage approach and templates improved separation of concerns. The model transformation is the same for all output languages and only output language specific code is defined in the templates. The result is that the size of the reimplementation is almost halved with respect to the original implementation of NunniFSMGen. We expect that the more strict separation of concerns reduces the work for adding a new output language than adding a new output language in the original implementation, where the model transformation also has to be reimplemented. On code level, the use of templates results in better readable object code, since object code is not embedded in println statements, which obfuscate the code by splitting it in substrings.

Beside the improved maintainability, the use of grammars and the use of the syntax safe template evaluator improve the correctness of the generated code of the reimplemented NunniFSMGen. Syntax errors are earlier detected in the reimplemented code generator, so users of the code generator are not confronted with syntax errors in the generated code. This is in particular an issue when the state machine is ported to a new output language. A transition table can contain valid syntax for the first language, but can be incompatible for the new output language. Finally, this case study shows that our templates can be used for generating code based on a behavioral pattern for different output languages.

7.7 Dynamic XHTML generation

The last case study we discuss is dynamic XHTML [96] generation in web applications. XHTML is a more restrictive version of HTML, so that it can be defined by a context-free grammar. This case study covers code generation during the usage of an application instead of using it for the development of the application. Beside just-in-time compilation [77], code generation during runtime plays an important role in contemporary applications communicating via the Internet. These web applications generate (X)HTML pages on request, which are interpreted by the browser to render the user interface of the application. Since the emergence of web applications a lot of web application frameworks, such as Java Spring Framework[11], Ruby on Rails[12], Django[13], PHP Zend[14], and so on, have been developed. These web application frameworks have in common that they are delivered with a kind of text-based template evaluator to render the (X)HTML output.

The problem of these web applications is to ensure that the fixed (X)HTML code in the templates does not contain syntax errors, which otherwise result in errors in the browser. More seriously is that dynamic code generation and the ability of browsers to execute code, like cascade style sheets (CSS) [16] and JavaScript [48], can result in security breaches. This case study shows the use of syntax safe templates to reduce the possibility of security bugs in applications generating code during runtime. We implemented a small shout-wall web application, where the security is enforced in a declarative manner by grammar definitions and syntax safe template evaluation.

We start with a discussion of cross-site scripting and how to prevent it. After that we present the implementation of the shout-wall web application and show the approach to ensure the web application is no longer vulnerable for cross-site scripting.

7.7.1 Cross-site Scripting

Cross-site scripting (XSS) is the class of web application vulnerabilities in which an attacker causes a victim's browser to execute untrusted JavaScript, CSS or (X)HTML tags with the privileges of a trusted host [123]. This untrusted code can collect data, change the look and/or change the behavior of the original web site. It is the number one of the top 25 security bugs in web applications in 2010[15]. Even contemporary large web-services, like Google,

[11] http://www.springsource.org (accessed on November 30, 2010)

[12] http://rubyonrails.org (accessed on November 30, 2010)

[13] http://www.djangoproject.com (accessed on November 30, 2010)

[14] http://www.zend.com (accessed on November 30, 2010)

[15] http://cwe.mitre.org/top25/ (accessed on May 25, 2010)

YouTube, Twitter and Facebook, have to deal with cross-site scripting. The main cause of cross-site scripting is the evolution of (X)HTML and the sub-languages can result in execution of code in the browser such as JavaScript and CSS. The way (X)HTML has evolved in the past decades resulted in a liberal interpretation of the (X)HTML language. Browsers, such as Firefox and MS Internet Explorer, even interpret (X)HTML code containing a lot of errors. Furthermore the evaluation function of the browser executes JavaScript or CSS embedded in the most exotic constructions, like hex encoding or code with embedded tabs or new lines[16].

The first requirement for a web application to be vulnerable for XSS is to have some untrusted (user) input, which is used for rendering the output. In a basic web application having a commit form and a result view this security bug is already present natively. An attacker can post some untrusted code between `script` or `style` tags in the commit form, which is interpreted by a browser rendering the result page.

A common architecture for these web applications is the model-view controller architecture as described in Section 7.2.3. The controller describes the behavior of the web application and is usually some application specific (business) logic written in a general purpose (script) language on top of a web framework, such as Django, Java Spring Framework or Ruby on Rails. The controller handles the web requests and returns the view to the web browser. The web framework performs web application domain specific tasks such as URL mapping, load balancing and so on. The information necessary for processing these web requests is stored in the model; usually implemented as a relational database. This database contains information loaded at deployment of the web application, submitted via the web or inserted via another external source. The view renders the web page and is most times implemented as a text template system querying the objects provided by the controller.

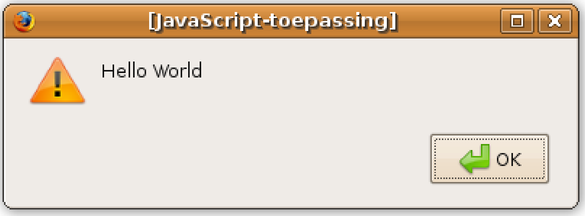
The problem of cross-site scripting arises when data is literally stored in the database and literally inserted in the rendered web page. An attacker can, for example, submit a piece of JavaScript

```
<script>alert("Hello World");</script>
```

on a reaction form on a web site. It is stored in the database and rendered on all the web pages of all viewers of that web site resulting in an annoying pop-up message, see Figure 7.38.

Most web frameworks already offer prevention against this naive form of cross-site scripting. The first solution is to scan the input committed via a web form before it is inserted in the database, so the database only contains *trusted* data. The problem of this approach is that it assumes that the database only contains trusted data. This is a valid solution as long as the database itself is

[16] http://ha.ckers.org/xss.html (accessed on July 20, 2010)


[JavaScript-toepassing]

Hello World

Figure 7.38 JavaScript pop-up message.

not vulnerable for attacks, but that is in practice not a realistic assumption. Besides that, alternative sources of data, such as RFID tags or URL parameters, are often seen by developers as trusted data and directly stored in the database, while they can contain malicious code [99]. At the end, we cannot trust any data stored in the database, and at the level of the web page generator we should consider all the data as untrusted to prevent cross-site scripting attacks.

The second solution against cross-site scripting provided by web frameworks is based on the assumption that the data necessary to render the web page can indeed not be trusted. Most attacks can easily be *disarmed* by replacing the characters < and > to a non executable equivalent, `&lt;` and `&gt;`, before inserting it in the final (X)HTML code. Contemporary template evaluators provided by web frameworks escape potential hazardous characters before the data is inserted in the web page by default. However, sometimes it is not feasible to use this escaping of characters as it is required to render layout information, like bold and italic tags, or it is even required to render a subset of JavaScript such as a JSON tree. At that point the escape mechanism must be turned off for that placeholder. As a result the web page generator is susceptible for cross-site scripting attacks.

In case character replacement cannot be used, a specific cross-site scripting filter or check can be defined. These filters are most times manually written in the general purpose language used for the controller component of the web application. These implementations contain a lot of unrelated details and the filter specification is scattered over the code. An example of a small hand-written filter is given in Figure 7.39[17]. It removes `<script>` tags, `javascript:` calls and `onXxxxx` attributes, like `onLoad` or `onClick`.

7.7.2 Preventing Cross-site Scripting

We present a solution for preventing cross-site scripting using syntax safe templates. In short, our solution uses syntax safe templates to parse the XHTML

[17] http://www.rgagnon.com/javadetails/java-0627.html (accessed on September 23, 2010)

```
public static String sanitize(String string) {
  return string
      .replaceAll("(?i)<script.*?>.*?</script.*?>", "")
      .replaceAll("(?i)<.*?javascript:.*?>.*?</.*?>", "")
      .replaceAll("(?i)<.*?\\s+on.*?>.*?</.*?>", "");
}
```

Figure 7.39 Hand-written cross-site scripting filter.

web-page including placeholders. The object code is already checked for well-formedness with respect to the XHTML grammar. The placeholders in the template are typed with the object language nonterminal they represent. This object language nonterminal is used to check that these placeholders are replaced by a valid (sub)set of the XHTML language during rendering the web page. Most times the language produced by the nonterminal of the placeholder is too broad to prevent cross-site scripting, as XHTML is defined as a language where between its tags almost the complete XHTML language is available. We introduced in Section 5.3.1 explicit syntactical typing of placeholders, which can be used to limit the language the placeholder can produce. When this subset of the XHTML language produced by the nonterminal of the placeholder is disjoint from the set of browser executable code; the syntax safe evaluator prevents inserting malicious code in the generated XHTML page.

Our solution to prevent injection of malicious code in the XHTML is based on filtering the data before it is inserted. Filtering can be based on the principle of *black-listing* or *white-listing* [123]. In case of white-listing, the set of allowed sentences is specified, in case of black-listing every sentence is allowed except the harmful ones, which are rewritten or removed by the filter. The problem of filtering is that browsers handle XHTML liberally and not in a uniform way, so covering the prevention of all manners of triggering the JavaScript and CSS evaluator is hard and different per browser. Wassermann et al. [123] have reviewed the source code and documentation of common browsers to obtain a list of ways to write executable JavaScript or CSS code. This list depends on the inspected version of the browser and for closed-source browsers this list is probably not complete. Without knowing which sentences are triggering the browser evaluation engine, it is hard, if not impossible, to ensure that cross-site scripting is prevented.

We advocate that a white-list system is preferable over a black-listing approach. White-list filters only include trusted syntax instead of excluding untrusted syntax. In case of a black-list filter there is always a chance that some untrusted syntax is not excluded. A white-list filter does not allow more syntax than necessary. Furthermore, white-listing is less susceptible for browser updates, as new ways of expressing executable JavaScript or CSS are most likely not allowed, except if this new way is a subset of the white-listed sentences grammar. We believe that testing, verifying or even proving that a context-free grammar only produces a language without harmful sentences is more feasible than proving the completeness of black-list filters.

The white-list filter in our syntax safe template approach is based on a context-free grammar for XHTML. Our XHTML grammar definition is a strict implementation of the XHTML specification [96] and more strict than the HTML syntax accepted by most browsers. Using this grammar for the object language, it is precisely defined which language a nonterminal can produce. When this language of a nonterminal does not contain sentences resulting in triggering the JavaScript or CSS evaluator, it can be safely extended with placeholder syntax. The result of using syntax safe templates combined with this XHTML grammar is that cross-site scripting protection is handled by the template evaluator instead of by hand-written error-prone filters.

7.7.3 Example Web Application

We built a web application to demonstrate our solution for preventing cross-site scripting attacks. This web application is a “shout wall” or “guest book” where a visitor can post a message and a name. The following requirements are defined for this example web application:

◇ For a post, the message field is mandatory and the name field is optional.

◇ Both fields, name and message, must contain human readable text, XHTML tags are not allowed.

◇ Beside human readable text, the message field may contain a JSON tree between `script` tags.

The last rule, allowing JSON data in the message field is for demonstration purposes. This requirement makes it impossible to use a naive character replacement to prevent cross-site scripting. Instead of rewriting characters, it must be verified that only well-formed JSON, without any executable JavaScript artifacts, is inserted in the output code.

The “shout wall” web application is implemented using the model-view controller architecture. We start with the discussion of the implementation controller and model. The controller class is listed in Figure 7.40 and extends the Java *servlet* API [63]. A servlet class may respond to *HTTP* requests. For this application, persistent storage of the data is not required and thus the model is also declared in the controller class by the field `messages`, which is initialized as an empty list. The model is based on ATerms 2.6.3, the input data tree format used by Repleo. The ATermLibrary is used to ensure the ATerms are well-formed.

The controller class implements two methods of the Java servlet API. The first method handles the *get* requests and returns a web page based on an

Metric	Shout Wall
Lines of Code	168
Lines of Code (without blank lines)	149
Tokens	1,980
Alphanumeric tokens	649
Non-alphanumeric tokens	812
White space tokens	519
Average number of tokens per line	13.29
Average number of non-alphanumeric tokens per line	5.45

Table 7.4 Metrics of the Shout Wall web application.

instantiated template. The template evaluator is invoked when the `doGet` is called. During evaluation of the template it will throw an exception if a parse error occurs. This parse error is the result sentence in the input data for message or name, which is not in the language defined by the grammar. The latest added message is removed from the list if an exception occurs and the web page is rendered again with a flag to display an error message. This second template evaluator call in the catch block is not enclosed in a try-catch statement, because the web application returns in a valid state when the last added message is removed. The web application is started in a valid state, i.e. a well-formed template and empty list of messages, only adding messages to the list can result in an exception, so it is always the last added message causing the exception.

The second method in the controller class handles the *post* request. It collects the data from the post request and stores it in a message object. The `ATmake` method of the ATerm factory is used to construct the message object to prevent ATerm injection.

The controller class does not contain any specific logic to prevent HTML injection attacks. Data committed by the user is directly stored in the model without any filtering. The only behavior responsible for the protection against cross-site scripting is the try-catch block in the controller. The controller expects that the template evaluator discovers the attack resulting in a generated exception.

The last discussed component of the "shout wall" web application is the view. The requirement for the view is that it only returns a well-formed XHTML page, otherwise it must throw an error. The template for the "shout wall" is shown in Figure 7.41. It contains XHTML code for the submit form and two match-replace placeholders; one for displaying an error message and one for rendering the messages list. The generation of this messages list is potentially vulnerable for cross-site scripting since substitution placeholders are used. Table 7.4 shows the metrics of the two files of the shout wall web application.

```
import java.io.*;
...

public class XssServlet extends HttpServlet {
  private ATermList messages;
  private aterm.pure.PureFactory termFactory = null;
  private WebPageGenerator pagegenerator;

public XssServlet() {
  termFactory = SingletonFactory.getInstance();
  pagegenerator = new WebPageGenerator();
  messages = termFactory.makeList();
}

public void doGet(HttpServletRequest req,HttpServletResponse res)
        throws ServletException, IOException {
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();
  ATerm inputdata = termFactory.make(
    "data(error(noerror),messages(<term>))", messages);
  String html = "";
  try {
    html = pagegenerator.generate(inputdata);
  } catch (Exception e1) {
// remove evil message from messages
    this.messages = messages.getNext();
    inputdata = termFactory.make(
      "data(error(detected),messages(<term>))", messages);
    html = pagegenerator.generate(inputdata);
  }
  out.println(html);
  out.close();
}

public void doPost(HttpServletRequest req,HttpServletResponse res)
    throws ServletException, IOException {
  String message = req.getParameter("message");
  if (!message.trim().equals("")) {
    String name = req.getParameter("name");
    ATerm messageNode =
      termFactory.make("message(<str>,<str>)", name,message);
    this.messages =
      termFactory.makeList(messageNode, this.messages);
  }
  this.doGet(req, res);
}
}

}
```

Figure 7.40 Java Controller of the “shout wall” web application.

```
template[
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Cross-site scripting prevention example</title>
</head>
<body>
<form action="/XssServlet.java" method="post">
<h1>Write your message on the wall</h1>
<br /><br />
<b>Message (PCDATA or JSON):</b><br />
<textarea name="message" cols="50" rows="7" class="textbox">
</textarea>
<br /><br />
<b>Name (only PCDATA):</b><br />
<input type="text" name="name" size="48" class="textbox">
<br /><br />
<input type="submit" value="Submit" class="textbox">
<br /><br />
</form>

<: match data1error1 sort:XHtml-flow-item* :>
 <: detected =:>
 <b style="color:red" >You tried to post some forbidden text!</b>
<br /><br />
 <: noerror =:>
<: end :>

<: match data2messages1 :>
 <: [ message($name, $text) , $t ] =:>
  <b>message:</b><br />
  <p><: $text sort:PCDATA-JSON :> </p> <br />
  <b>name:</b> <i><: $name sort:PCDATA* :></i> <br /><br />
   <: $t :>
 <: [ ] =:>
<: end :>

</body>
</html>
]
```

Figure 7.41 XHTML template of the “shout wall” web application.

The view is implemented using syntax safe templates, as a result the template is parsed and the placeholders have a syntactical type. This syntactical type is used by the evaluator to substitute the placeholders only with sentences belonging to the language of that syntactical type, otherwise an error is generated. This behavior of syntax safe template evaluation is used to provide the protection against cross-site scripting attacks. An error during the evaluation process indicates that the input data contains a sentence which is not allowed to replace the substitution placeholder.

The requirements state that the message may contain human readable text or a JSON tree and that the name is optional and must be human readable text. Human readable text without XHTML tags is provided by the XHTML grammar as the `PCDATA` nonterminal. PCDATA may contain every character

```
module JSON

exports
  sorts JSON JSONTUPLE JSString JSStrChar

  context-free syntax
   "{" { JSONTUPLE  "," }* "}"   -> JSON
   "[" { JSON "," }* "]"         -> JSON
   JSString                      -> JSON
   JSString ":" JSON             -> JSONTUPLE

  lexical syntax
    ~[\0-\31\n\t\"\\\<\>]        -> JSStrChar
    [\"] chars:JSStrChar* [\"] -> JSString
```

Figure 7.42 JSON Grammar definition.

except the characters <, > and &. We enforced the substitution placeholder responsible for generating the name as `PCDATA*`, where the star list sort allows to substitute it with no text. This protection against cross-site scripting attacks is easy to implement and indeed already supported by most web template systems. They provide automatic escaping for the three forbidden characters.

In order to show that syntax safe templates are more advanced than simple escaping of some characters in a string, we required that the text of the message may contain human readable text *or* a JSON tree. Escaping dangerous characters cannot be used anymore as JSON trees between script tags contain these characters. A grammar must be defined for JSON only producing valid JSON sentences not resulting in execution in the browser. We defined a JSON grammar in SDF, see Figure 7.42, conforming to the JSON standard [35]. Our JSON grammar defines a subset of the original JSON language specification by limiting the class of characters allowed for `JSStrings`. The greater-than and smaller-than characters are explicitly disabled, otherwise it is possible to embed a cross-site scripting attack inside a JSON tree.

JSON is not a part of the XHTML grammar, thus we mixed both languages by defining the grammar module of Figure 7.43. This grammar module adds the nonterminal `PCDATA-JSON` as alternative to the `XHtml-Inline` of the XHTML grammar. The `PCDATA-JSON` nonterminal produces the language of sentences containing the set of JSON trees between `script` tags and the set of sentences provided by `PCDATA*`. The production rule adding `PCDATA-JSON` to the XHTML grammar is annotated with `avoid` to specify that it should never be used if another production rule can be applied. The priority of the added nonterminal `PCDATA-JSON` is lower than the original alternatives for `XHtml-Inline`.

We specified the substitution placeholder for the text part message explicitly as the syntactical type `PCDATA-JSON`. The template evaluator is allowed to replace it by human readable text *or* JSON trees. The protection against JavaScript and/or CSS injection is further handled by the syntax safe template evaluator

```
module XHtml–Json

imports XHtml
imports JSON

exports
  sorts PCDATA–JSON
  context–free syntax
    PCDATA–JSON              –> XHtml–Inline {avoid}
    PCDATA*                  –> PCDATA–JSON
    "<script" ">" JSON "</script>"
                             –> PCDATA–JSON
```

Figure 7.43 Adding JSON as alternative for XHtml-Inline.

using the grammar.

7.7.4 Practical Validation

In order to verify the *unhackability* of the shout wall, we published the shout wall website including the source code and asked a number of web security experts to find security breaches in the implementation. Two bugs were found in the first shout wall implementation. First, Hayco de Jong discovered that an input term containing a repeating pattern `{ "a",` caused a crash of the webserver. This was a result of a stack overflow in the parser and is solved by fixing the stack overflow handling in the Java–to–C connection of the SGLR parser.

The second security issue, discovered by Aram Verstegen, was a cross-site scripting problem. The problem discovered by Aram is inherent in the JSON language specification [35], which allows strings of the character class

```
~[\0-\31\n\t\"\\].
```

This character class allows the symbols < and >, JavaScript encapsulated in a string is valid conform this grammar, but Firefox 3.0.19 interprets this as JavaScript code stored in JSON strings instead of handling it as data. We solved this security breach by limiting the character class for `JSStrChar` in Figure 7.42 by disallowing the symbols < and >. After these two errors were detected, no further security issues were found.

As discussed, the use of grammars in a syntax safe templates does not prevent all injection attacks. However, having formalisms for grammar definitions and syntax safe templates provides a compact and understandable implementation of web applications. The use of languages at a higher level of abstraction, such as grammar definitions, prevents making silly errors resulting in security breaches. It is easier to examine the allowed sentences of a grammar than inspecting manually written parsers and filters. When a security breach is found, it is easier to repair in a grammar, since the formalism has a declarative nature.

7.7.5 Related Work

A similar solution to prevent cross-site scripting and/or injection attacks is *syntax embeddings* presented by Bravenboer et al. [25]. This approach prevents injection attacks in sentences of an embedded language manipulated in some host language. It uses a grammar based on the host language extended with syntax of the embedded language to achieve the safety. An API is generated from this grammar to provide the appropriate unparsing, escaping, and checking of lexical values. The source code containing embedded syntax is translated to a file without embedded syntax by replacing this foreign syntax with calls to the generated API. Although our approach of preventing injection attacks is based on the same principle of combining grammars as suggested in their approach, there is a difference in the application of it. The application domain of syntax embeddings is to increase printf based code generators and replace the printf statements by object language artifacts. In contrast to our approach, where injection attacks are prevented in the context of template based code generators. This manifests itself by the fact that the grammars used for syntax embeddings are based on a metalanguage grammar where object language nonterminals are injected as alternatives, while in template grammars the object language grammar is extended with metalanguage constructs.

7.7.6 Evaluation

We already mentioned that cross-site scripting attacks are nowadays the number one security bug in web applications. We showed that syntax safe templates can provide a solution to prevent cross-site scripting without introducing a lot of boilerplate and checking code. Grammars are used to specify allowed (sub) sentences in the template instead of implementing the checks and filters for the input data in the controller component. Filters and checks implemented in a general purpose language contain a lot of detail, which make it hard to write, to maintain, to test and to validate these manually implemented filters and checkers, especially for complex languages. Obviously, also in our case the level of protection against injection attacks is dependent on the quality of the used grammars. However a context-free grammar definition is easier to maintain than checkers implemented using imperative constructs.

Beside the improved security during evaluation, parsing both XHTML and metalanguage provides easier development of these templates. We used an IDE for editing syntax safe templates, which reports syntax errors directly in the editor. Syntax errors in the object code and meta code are already detected before any web-page is generated. Syntax safe templates can of course be used for preventing injection attacks when other output languages than XHTML should be generated dynamically.

7.8 Conclusions

We discussed four case studies, targeting different application areas of code generation. The case studies were chosen to validate that the metalanguage in combination with the two-stage architecture results in less code, better maintainable, and to show the benefits of syntax safety. For the first three case studies, using code generation for application development, the use of a two-stage architecture results in a better separation of concerns between the model transformation and code emitters.

The compact metalanguage, as designed in Chapter 4, enforced us to use the two-stage architecture. It is not possible to express all model transformations in our metalanguage, as it is not possible to express calculations and store intermediate values. This compact metalanguage complies with the recommendation of Parr [92] to define a metalanguage to enforce strict separation of model and view. Since the metalanguage is unparser-complete, it is not a limitation of the code the templates can instantiate. This is empirically validated by the four presented case studies.

The templates do not contain complex calculations on the model and the model transformations do not need to contain object code artifacts. Especially in the two case studies, ApiGen and NunniFSMGen, reimplementing a code generator, the benefits of the two stage architecture based on term rewriting and templates are reflected in the reduced number of lines of code. Term rewriting provides a powerful computational paradigm to express the tree transformations used in the model transformation stage, while the templates only consider the output code rendering. The reimplemented code generators are at least half the size of the original implementations. We showed in the NunniFSMGen case study that the choice of the output language is made at the template level, while the input data model is not changed. We expect that the increased separation of concerns in the reimplementation results in easier adding a new output language to our NunniFSMGen implementation than adding a new output language to the original implementation.

Table 7.5 shows the metrics of all (re)implemented code generators. Considering the size metric, WMSL is the biggest case study and the shout wall application the smallest. More interesting is the average number of (non-alphanumeric) tokens per line metric. The number of tokens per line is between 13.29 and 15.4, and the number of non-alphanumeric tokens per line is between 5.45 and 6.64. Considering the metrics of the original implementation of ApiGen (see Table 7.2), the number of tokens per line is 11.5 and the number of non-

Metric	WMSL	ApiGen	NunniFSMGen	Shout Wall
Lines of Code	2,956	2,975	1,005	168
Lines of Code (without blank lines)	2,435	2,361	738	149
Tokens	34,638	33,254	10,438	1,980
Alphanumeric tokens	9,792	7,572	2,393	649
Non-alphanumeric tokens	14,483	15,669	4,762	812
White space tokens	10,363	10,013	3,283	519
Average number of tokens per line	14.23	14.08	14.14	13.29
Average number of non-alphanumeric tokens per line	5.95	6.64	6.45	5.45

Table 7.5 Metrics of the different case studies.

alphanumeric tokens per line is 4.62. Future work is to investigate the meaning of these numbers. What does the alphanumeric tokens and non-alphanumeric tokens ratio say about the readability of a piece of code? Is it possible to categorize code based on these metrics, i.e. is the non-alphanumeric tokens per line between 5.45 and 6.64 typical for code generators?

The role of syntax safety is not good quantifiable in the first three case studies, as syntax safety directly warns a developer during writing templates in the IDE. During implementing the templates we experienced these error messages as very helpful. Furthermore, these syntax errors during code generation are scarce, if not impossible, as we used model transformations based on a term rewriting system requiring that the input and output are sentences of a predefined grammar. However, the fourth case study shows that syntax safety increases the safety of dynamic code generation in web applications. Cross-site scripting is prevented without introducing a lot of boilerplate and checking code. Grammars are used to specify allowed (sub) sentences in the template instead of implementing the checks for the input data in the controller component.

Syntax safe templates as presented in this thesis can be used in the different application areas of the discussed case studies without any adaptation. This shows that syntax safe templates are applicable in real world code generator cases.

8

Towards Static Semantic Validation

Based on our previous work on detection of syntactic errors in Chapter 5, in this chapter we study techniques to detect static semantic errors in templates in order to prevent static semantic errors in the generated code. Static semantic checks are defined for the metalanguage, for the object language and checks for the situations where the object language is dependent on the metalanguage. Moreover, we investigate the possibility of a re-usable approach: in the same way the template syntax extends the syntax of a programming language, our approach extends a static semantic checker for the corresponding programming language. An implementation of a static semantic checker for PicoJava templates is discussed. The requirements for a re-usable static semantic checker are shown based on the experiences of implementing the static semantic checker for PicoJava templates. We conclude with formulating research questions for future work.

8.1 Introduction

Based on our previous work on detection of syntactic errors in Chapter 5, in this chapter we study techniques that go beyond the syntactic level. We want to detect semantic errors in the template *prior* to its use for generation of code. In this chapter we discuss our ideas for implementing semantic checkers for templates.

We start with a discussion of static semantics of programming languages. Next, an overview of static semantic checks for templates is given. After that, we discuss our prototype of a static semantic checker for templates based on attribute grammars. Finally related work, conclusions and future work are presented.

8.2 Static Semantics

A context-free grammar defines the syntax, i.e. a set of sentences belonging to a language. The semantics of a language refers to the *meaning* of these sentences, which must be preserved by compilers or other language implementations [91]. Following [91], the semantics of a language can be split up in a static part and an operational part. The operational part describes the execution behavior of program language constructs, while the static part describes restrictions on the structure of the text of a program, which cannot be expressed using a context-free grammar. This *static semantic* checking considers, amongst others, the declaration of variables, use of variables and their types, and it manages the typing semantics, which considers the types of all program phrases. Examples of checks are detecting redefinitions of an identifier in the same scope or detecting type errors such as the assignment of a value to a variable of the wrong type. Static semantic checking is sometimes called type checking, which is in fact a part of the static semantic checker. Implementing static semantic checkers for compilers is discussed in [5] and static semantic checkers for homogeneous meta-programming languages are discussed in [110].

Recall that a template is a sentence of a template language, which is an extension of an object language. Original object language tools, such as its static semantic checkers, cannot be used to process these templates. In case of the heterogeneous template languages, as discussed in Section 5, the approaches of Aho et al. [5] and Taha et al. [110] for implementing a static semantic checker are not directly applicable. The approach of Aho et al. is aimed at implementing static semantic checkers from scratch. The authors discuss implementations of static semantic checkers based on directly embedding the checker in the parser definition, not particular intended for re-usability, as the parsing and checking are mixed in one stage [18]. The approach of Taha et al. is designed in the context of homogeneous meta-programming. It is based on the assumption that metalanguage and object language are identical and both languages use the same type environment.

The aim of this chapter is to implement a static semantic checker for template languages based on an arbitrary object language. We define two explicit requirements:

- ◇ The reported errors are sound.
- ◇ The approach should re-use as much as possible of semantic checkers and grammars of existing object language implementations.

Furthermore we aim for detecting as many errors as possible in a template, but the approach does not necessarily need to find all the errors, i.e. being complete. The proposed approach is not intended to statically guarantee that a template always generates semantically correct code, but we consider extra checking and every detected error useful for preventing bugs in the generated code. In the next section we discuss our ideas to validate the static semantic properties of templates.

8.3 Defining Static Semantic Checkers

As we already discussed in the previous chapters, the syntax of programming languages can be specified using formal grammars (see Section 2.2). Based on the theory of formal grammars, formalisms, such as BNF [1] and SDF [55], are developed to define syntaxes. They are widely used to define the syntax of programming languages and to generate parsers from them.

As opposed to the case of syntax, there is no widely accepted standard for describing the semantics [91]. The literature provides approaches to describe the semantics of programming languages in a formal way. Examples are modular structural operational semantics [85] and *attribute grammars* [73], which we used for our implementation. However, the semantics of industrially used programming languages are most times specified in an informal way [91]. For example, the specification of the languages C [64] and Java [49] consists of English sentences. It is hard verify whether a description in the English language covers every detail and is free of ambiguities. For instance, in the description of the Java language a couple of semantic ambiguities have been found [28]. When using different compilers, a semantic ambiguity can result in different behavior or different errors.

In order to be usable, the specification of the semantics of a language must be implemented in tools like a static semantic checker, compiler or interpreter. Unfortunately, the lack of a standardized approach and accompanying formalisms to specify the (static) semantics of a programming language has the consequence that these tools are most times implemented in an ad-hoc way. The operational implementation can result in not well separated stages of checking, code optimization or generation, or in a situation where the operational implementation of the static semantic checker is not easy to extend with new rules. Re-using these implementations of static semantic checkers is thus hard.

In the same way a template grammar can be composed from an object language grammar and metalanguage grammar, we want to extend a static semantic checker for the object language grammar to handle the meta code. We have chosen to use *JastAdd* [54]. Two reasons underlie the choice of JastAdd. First, it claims to offer modular specifications of compiler tools [38]. Second, a working JastAdd based Java compiler implementation called JastAddJ is available [37].

The next section will discuss JastAdd. JastAdd is a contemporary Java-based implementation of an attribute grammar system. The formalisms provided by JastAdd are presented by means of the toy language *PicoJava*. The examples show how to define static semantic properties of a language. JastAdd does not include a parser and assumes that an external parser is used to instantiate an abstract syntax tree. SGLR is connected to JastAdd since we need this parser to use our SDF based template grammars. The connection between SGLR and JastAdd is discussed in Section 8.4.

8.3.1 Attribute Grammars with JastAdd

Attribute grammars, introduced by Knuth [73], are a formalism to describe context-sensitive properties of constructs in a language in a declarative manner. They are an extension of context-free grammars, where attributes can be attached to nonterminals and semantic rules to the production rules. An attribute is a property assigned to a nonterminal in the grammar and can contain values dependent on its context in the syntax tree. Attributes occur in two kinds: *synthesized attributes* and *inherited attributes*. Synthesized attributes are the result of attribute evaluation rules and are used to pass information upwards in the parse tree. Inherited attribute are passed down from parent nodes. A parse tree for a sentence in the language of an attributed grammar has the same form as a context-free parse tree. However, each node is additionally decorated with the values of the attributes of the respective symbols.

A contemporary Java based implementation of an attribute grammar evaluator is JastAdd [54]. JastAdd uses an object-oriented representation of the abstract syntax tree in combination with reference attribute grammars. Reference attribute grammars [53] are an extension of the original attribute grammars as defined by Knuth [73]. The problem of the original attribute grammars is that non-local dependencies are hard to specify, such as name analysis where properties of an identifier usage depend on properties of an identifier declaration. Information must be manually propagated through all the nodes of the tree via an *environment* attribute, where information can be stored and looked up. This can become very complex in languages with complex scope rule, like object-oriented languages. Besides that, the original attribute grammar is difficult to extend due to the propagation requirement of attributes. When adding new constructs to the language, all nodes should propagate the environment attribute. The result is that the environment attribute of the extended language should be added to the new nodes and environment attribute of the new nodes should be added to the nodes of the extended language. This introduces tangling of the added language constructs and the existing language.

Reference attribute grammars offer a solution, which does not need an environment attribute in every node of the syntax tree. This way, a reference attribute constitutes a direct link from one node to another node arbitrarily far away in the syntax tree. Information can be propagated directly from the referred node to the referring node, without having to involve any of the other nodes in the syntax tree.

```
{
boolean i;
i = i + 7;
i = i + 3;
}
```

Figure 8.1 Simple PicoJava program. Result of the template evaluated with `X([7,3])`

```
{
boolean i;
<: foreach $x in X1 do :>
  i = i + <: $x :>;
<: od :>
}
```

Figure 8.2 Template with a simple iteration.

JastAdd uses an abstract syntax definition and a reference attribute grammar module as input to create a language processing application. The abstract syntax definition is converted to Java classes which are used for instantiating the abstract syntax tree. The reference attribute grammar contains the equations and attributes for the production rules in the grammar. Aspect-oriented programming [68] is used to weave the attribute grammar into the generated classes. JastAdd does not provide a parser and expects that an external parser is used to create an abstract syntax tree from concrete code. The combination of object-oriented implementation of the abstract syntax tree and the weaving of the attributes into these classes enables modularization of the different components of a static semantic checker of a language.

8.3.2 PicoJava

We use a subset of Java, known as PicoJava as object language to illustrate JastAdd. PicoJava[1] is designed to demonstrate features of the attribute grammar system JastAdd [54] by means of an implementation of name resolution and type analysis for it. A PicoJava program consists of *blocks*, where blocks contain *statements*, *class declarations* and *variable declarations*. An example of a PicoJava program is shown in Figure 8.1. JastAdd is distributed with a static semantic checker for PicoJava. We have extended PicoJava to make more interesting examples. We included a conditional statement, integers and strings as basis types, and a few binary and unary operators. Since we are interested in PicoJava templates, an example of such a template is given in Figure 8.2.

Figure 8.3 shows a fragment of the PicoJava grammar. The constructor information is used to obtain an abstract syntax tree from a parsed PicoJava program, which is used for connecting SGLR to JastAdd. The `skip` constructors are implemented via an inheritance relation and not represented as nodes in the abstract syntax tree. We discuss the connection of SGLR and JastAdd in Section 8.4.

[1] http://jastadd.org/jastadd-tutorial-examples/picojava-checker (accessed on November 30, 2010)

```
module PicoJava/syntax/PicoJava
imports basic/Whitespace

hiddens
 context-free start-symbols Program

exports
 sorts Program Block BlockStmt ClassDecl VarDecl
       Stmt AssignStmt

 context-free syntax
  Block                    -> Program {cons("Program")}
  "{" BlockStmt* "}"       -> Block {cons("Block")}
  Stmt                     -> BlockStmt {cons("Skip_0")}
  Decl                     -> BlockStmt {cons("Skip_1")}
  TypeDecl                 -> Decl {cons("Skip_2")}
  VarDecl                  -> Decl {cons("Skip_3")}
  Access Identifier ";"    -> VarDecl {cons("VarDecl")}
  ClassDecl                -> TypeDecl {cons("Skip_4")}
  "class" Identifier ("extends" IdUse)? Block
                           -> ClassDecl {cons("ClassDecl")}
  AssignStmt               -> Stmt {cons("Skip_5")}
  Access "=" Exp ";"       -> AssignStmt {cons("AssignStmt")}
```

Figure 8.3 Fragment of the PicoJava SDF definition.

8.3.3 Abstract Syntax Definition

The abstract syntax definition is a formalism to specify the regular tree grammar belonging to the abstract syntax trees of a programming language. JastAdd translates these abstract syntax definitions to a Java implementation of the interpreter pattern, as discussed in Section 7.5.7. The interpreter pattern implements an abstract syntax tree using a class hierarchy, where nonterminals are abstract super classes and alternatives are concrete classes inheriting the accompanying abstract super class. For each nonterminal in the abstract syntax definition a class is generated with accessor methods.

The abstract syntax definition consists of the definition of nonterminals and production rules. The production rules of the abstract syntax definition have the following form:

```
<nonterminalName> : <inheritsFrom> ::= <productionRule>;
```

The `nonterminalName` will result in a new class `nonterminalName`, which may inherit from another class (i.e. nonterminal). The production rule defines the children of a node and may be empty. If non-empty, the production rule has the following form:

```
... ::= <label1>:<nt1> <label2>:<nt2> ...;
```

```
Program ::= Block;
Block ::= BlockStmt*;

abstract BlockStmt;
abstract Stmt: BlockStmt;
abstract Decl: BlockStmt ::= Identifier;
abstract TypeDecl:Decl;

ClassDecl: TypeDecl ::= [Superclass:IdUse] Body:Block;
VarDecl: Decl       ::= Type:Access Identifier;
AssignStmt: Stmt    ::= Variable:Access Value:Exp;
```

Figure 8.4 Fragment of the abstract syntax for PicoJava.

The label is used to identify a nonterminal field. Nonterminals can be specified by the keyword `abstract` when only derived nonterminals are allowed to instantiate.

Figure 8.4 shows a part of the abstract syntax of PicoJava. The names of the nonterminals match the names used in the constructors of Figure 8.3. These constructor names have all an accompanying production rule in JastAdd, except for the productions with the constructor name of the form `Skip_x`. These rules are not present in the abstract syntax definition. When an abstract syntax tree is created by the parser, the nodes are labeled with the corresponding constructor name. These nodes are related to the production rules in the abstract syntax definition. The `Skip_x` rules reflect nodes that represent chain rules specified in the SDF grammar, but are not necessary in the abstract syntax tree used by JastAdd, since they can be intercepted by an inheritance relation. For example the inheritance relation of `Stmt` is a subclass of `BlockStmt`.

8.3.4 Attributes

The abstract syntax definition is used to define the regular tree language of the abstract syntax trees. In order to define compilers and static semantics checkers, the abstract syntax definition needs to be extended with behavior. The generated interpreter pattern can be extended with attributes and equations via aspect-oriented programming (AOP) [68]. This AOP approach enables a modular way for defining the semantics of the language.

The attributes are defined separately from the grammar. Three main modifiers for these attributes are available. The inherited modifier, `inh`, is used for passing information down the tree, the synthesized modifier, `syn`, is used to pass information up the tree. Equations are written like Java assignment statements preceded by the `eq` modifier. As mentioned, the abstract syntax definition of JastAdd uses inheritance; attributes and equations added to a superclass of a nonterminal are available to its subclasses. After classes are

```
aspect TypeAnalysis {
 syn TypeDecl Decl.type();
 syn TypeDecl Exp.type();
 ...
 // ***Implementation***
 eq TypeDecl.type() = this;
 eq VarDecl.type() = getType().decl().type();
 ...
 eq BooleanLiteral.type() = booleanType();
 ...
 eq BinOperator.type()  {
  if (getlhs().type().isSubtypeOf(getrhs().type())
   || getrhs().type().isSubtypeOf(getlhs().type()) )
   {
 // special case: if the type of either of them is
 // unknown, the known one should be the returned type
    if (getlhs().type().isUnknown() )
      return getrhs().type();
    return getlhs().type();
   }
  return unknownDecl().type();
 }
}
```

Figure 8.5 A fragment of the type rules.

generated using the abstract grammar, these attributes are incorporated into these classes.

A part of the static semantic checker for PicoJava is shown in Figure 8.5. It shows the synthesized attribute `type` for the classes `Exp` and `Decl` with the type `TypeDecl`. An attribute gets its value from an equation, i.e. the attribute `type` gets its value from the equation belonging to the specific class. Equations are shown for the classes `TypeDecl`, `VarDecl`, `BooleanLiteral` and `BinOperator`. The equation for `TypeDecl` and `BooleanLiteral` directly return the type of the node. The `VarDecl` returns the type defined by its declaration and `BinOperator` calculates its type based on the type of its operands.

8.4 Connecting SGLR and JastAdd

A parser is needed to get an instantiation of an abstract syntax tree. JastAdd does not contain a parser and requires an external parser. In our template evaluator we use the SGLR parser [117]. In this section we discuss how to connect the SGLR parser to a JastAdd based static semantic checker.

SGLR generates a parse tree when it parses a text. An instantiation of the abstract syntax tree in JastAdd is created by traversing this parse tree of SGLR. By traversing the tree the constructor name defined in the SDF grammar is used to select the JastAdd class via reflection. Therefore the constructor names in SDF must match the nonterminal names in the abstract syntax definition of

```
public void ASTNode.collectErrors(Collection c) {
  for(int i = 0; i < getNumChild(); i++)
    getChild(i).collectErrors(c);
}
```

Figure 8.6 The object language error collection method.

JastAdd, otherwise a more complicated mapping scheme is necessary. There is one special type of constructor, which is the `Skip_x`. A node with this constructor is ignored by the traversal. The traversal continues with the children of the skip node, but no JastAdd tree object is instantiated for this skip node.

JastAdd is delivered with a static semantic checker for PicoJava. We connected the SGLR parser to this JastAdd based static semantic checker. This allows us to check PicoJava programs using SDF as grammar formalism.

8.5 Static Semantic Checks for Templates

In a template, we have the metalanguage and the object language fragments executed at different stages. The metalanguage is interpreted by the template evaluator, while the object language is handled as data. The object language defines the language of the code (fragments) to compose the output, which is compiled or interpreted by an external tool. These different execution stages reflect in the different levels of static semantic checks. We can distinguish three levels of checks in a template:

◇ Metalanguage checks;

◇ Object language checks;

◇ Object language checks dependent on the metalanguage.

In this chapter we investigate the possibility to detect static semantic errors in templates, where we use PicoJava as example object language. We extend the PicoJava static semantic checker provided by the JastAdd project in order to accept PicoJava templates [54]. The structure of the interpreter design pattern of the AST allows us to declare these checks independently of each other. Every class inherits from a generic node class, called `ASTNode`. Using aspect-oriented programming, the node class is extended with three error collecting methods, one for each kind of checks: `collectErrors`, `collectIndepErrors` and `collectPhErrors`. The `collectErrors` method for the static semantic errors of the object language is shown in Figure 8.6.

The classes implementing the `ASTNode`, i.e. the nonterminal classes, may override the error collecting method to implement a static semantic check. This mechanism allows us to mix different static semantic checkers for different languages inside one tree, since a node not overriding the function executes the default recursive behavior of the error collecting method.

In the rest of this section we will discuss these checks and give their implementation in JastAdd. For instance, by inspecting the template in Figure 8.2 we would like to detect that the attempt of adding integers to a boolean value will cause a type error when compiling or interpreting the generated code (Figure 8.1, line 2). Since JastAdd is distributed with an implementation of a static semantic checker for PicoJava, we will use PicoJava as object language for our templates. The first step is done by connecting the PicoJava templates syntax definition to JastAdd. Establishing a connection between the parser and the static semantic checker provides already out-of-the-box checking of the object code of the template without placeholders, but placeholders are still not handled. More details about the implementation of the checker for PicoJava templates can be found in [95].

8.5.1 Metalanguage Checks

The metalanguage is independent of the object language and has its own static semantic properties. A static semantic checker can be defined for the metalanguage. This checker ignores the object code and only considers the static semantic properties of the metalanguage.

In case of our metalanguage, as defined in Chapter 4, two levels of semantic checks are possible. The first level contains the checks validating that the meta code itself does not contain static semantic errors. The second level contains the validation whether the described match-patterns belong to the regular tree grammar of the input data. When this is not the case, it can lead to match-pattern evaluation errors.

Our metalanguage is small, so we define a short, not necessarily complete, list of semantic rules:

1. Identifiers of (sub)templates must be unique.
2. Meta-variables in a match-pattern must be unique.
3. The `$root` meta-variable is not allowed in a match-pattern.
4. Used meta-variables must be assigned in a parent match-pattern.

Rule 1 eliminates the possibility that more than one (sub)template can be selected by a subtemplate invocation. Without this rule it is possible to define multiple subtemplates with the same identifier resulting in nondeterministic subtemplate selection. Observe that in a syntax safe template setting, it is possible to use the same identifier for subtemplates having different root nonterminals. The root nonterminal of the subtemplate can be used to distinguish between the different subtemplates having the same identifier. However, for readability, we prefer not to allow this option, since selecting a subtemplate based on its root nonterminal is not explicit.

Rule 2 eliminates multiple declarations of a meta-variable in a matching-pattern. It is not allowed to redefine a meta-variable in the same scope. Our symbol table mechanism (see Section 4.3) does not allow the same key multiple times in a single scope. Besides that, in functional programming it is not unusual to support *non left-linear matching* [83], i.e. meta-variables with the same identifier must match against the same sub patterns to result in a successful match. While our metalanguage is not designed to behave in this way, we still want to prevent possible confusion and, hence enforce uniqueness of meta-variables in a match-pattern.

Rule 3 checks whether the `$root` meta-variable is assigned in a match-pattern. This meta-variable is reserved for the template evaluator and it is assigned to the root of the original input data by the template evaluator. Overriding this meta-variable in a match-pattern will break the intention of the `$root` meta-variable to contain the original input data tree.

Rule 4 requires that every meta-variable is assigned in a parent match-pattern before it is used. If a meta-variable is not assigned, the template evaluator should return an error.

While the semantic rules discussed above can be used to check the meta code without external information, the following checks use the input data tree grammar to validate the meta code. The meta code of a template describes, via the match-patterns and tree path queries, a kind of tree automaton accepting input data trees. This automaton should at least accept all the trees defined by the input data tree grammar. In case the input data language produces more trees than the template accepts, there is a set of input data trees resulting in an error during template evaluation. The following errors should be detected using the tree path queries, match-patterns and input data grammar to verify that the template accepts the input language:

5. A tree path query does not specify a valid path for the nonterminal n.

6. The match-pattern does not match any tree produced by nonterminal n.

7. The match-replace placeholders should contain a match rule for every alternative of a nonterminal n defined in the input data grammar.

The fifth rule and the sixth rule check whether the tree path query or match-pattern can match a (sub)tree belonging to the input data language defined by its regular tree grammar. The checks detect tree path queries which do not define a valid path, resulting in an error during evaluation, and detect useless match-patterns.

```
public void PhIdentifier.collectIndepErrors(Collection c) {
  super.collectIndepErrors(c);

  if (! getMatchVar.matchvars().containsAll(
          getIdentifier().getTreeQuery().matchvars() ) ) {
    Collection remainder = getIdentifier().matchvars();
    remainder.removeAll(getMatchPattern().matchvars()) ;
    error(c, "the meta-variables "+ remainder.toString() +
          " have been used that were not declared");
  }
}
```

Figure 8.7 Collecting undeclared meta-variables.

A match-replace placeholder defines match-rules for a nonterminal *n* of the input data grammar. The seventh rule checks whether a match-replace has a match-rule for every alternative of the nonterminal *n*. In case a match-replace placeholder does not cover all possible patterns of nonterminal *n*, it is possible that a (sub)tree in the input data has no accompanying match-rule, resulting in an evaluation error.

Figure 8.7 shows the implementation of Rule 4. This rule requires to check whether a value is assigned to the given meta-variable. If the meta-variable is not assigned, an error is reported. Verification is performed by collecting the declared meta-variables and checking in the placeholders using the meta-variables whether they are available in this set. An error is generated when a meta-variable is not present in the declared collection.

This check method overrides `collectIndepErrors` defined in the class `ASTNode`, and thus called by this error collection traversal. Object language nodes are ignored by the traversal, since these node classes do not override the `collectIndepErrors` method. Scoping is automatically solved, since the collection is extended with declared meta-variables when the traversal descends in the tree.

8.5.2 Object Language Checks

The specification of the object language should provide its context-free grammar, the static semantics and operational semantics. Often a compiler or interpreter for the object language implements the static semantic checks. Examples of object language checks are uniqueness of variables checks and type checks [5]. In case a template has no placeholders, it is a sentence of the object language. In that case the object code can be checked by an existing static semantic checker for the object language.

However, in case a template contains placeholders the static semantic checker of a compiler or interpreter cannot be used, as it has no support for the

```
...
// Identifier
PhIdentifier        : Identifier ::= TreeQuery;
PhIfIdentifier      : Identifier ::= TreeQuery MatchPattern
                         Then:Identifier Else:Identifier;

// BlockStmt*
PhBlockStmt         : BlockStmt  ::= TreeQuery;
PhIfBlockStmt       : BlockStmt  ::= TreeQuery MatchPattern
                              Then:BlockStmt*  Else:BlockStmt*;
PhForeachBlockStmt: BlockStmt  ::= MatchVar TreeQuery
                              BlockStmt*;
...
```

Figure 8.8 Fragment of the abstract syntax for the placeholders of PicoJava.

placeholders. To handle templates, the static semantic checker needs to be extended with the semantic properties of the metalanguage. What does it *mean* to have placeholders in object code? In case a placeholder is added to the object language grammar, it not only inherits the syntactical properties of the object language nonterminal, but also its semantic properties. Since the final value of the placeholder depends on the input data, the use of placeholders will cause disturbances within the semantics of the object language. Extra rules must be defined to specify the meaning of a placeholder in that part of the language. We show a solution based on inheritance to add the semantics of an object language nonterminal to a placeholder nonterminal.

The static semantic checker for the language PicoJava is already provided by the JastAdd project. It has support for basic name resolution, dot name resolution, inheritance name resolution, and type analysis. Before the PicoJava checker can handle the added template constructs, the placeholder nodes must be added to the abstract syntax definition of the object language. Placeholders are added to the context-free grammar of the object language via injections (chain rules), at the abstract syntax level this is done via inheritance. Beside the syntactical properties, placeholders also inherits the semantic properties of the nonterminals of the object language. Indeed, a placeholder acquires all features the corresponding object language nonterminal possesses and adds specific features pertaining to the metalanguage. This allows us to re-use an existing static semantic checker by inheriting the placeholders from an object language nonterminals in the abstract syntax definition. Figure 8.8 shows the definition of the placeholders. We have to instantiate every placeholder manually for each nonterminal extended with placeholder syntax, just as the definition of a template grammar.

At the moment a placeholder inherits an object language nonterminal, it not only inherits it syntactical structure, but also its semantic properties. Since a placeholder does not have a value, but is a hole, the use of placeholders will cause disturbances within the semantics of the object language. It is necessary

```
aspect PhNameResolution {
  eq PhIdentifier.getName() =
            "<: " + getTreeQuery().value() + " :>";
  eq PhIdentifier.checkName(String name) =
            name.equals(getName());
}
```

Figure 8.9 Extra equations for substitution placeholder.

to override the definition of the attributes such that the object language checker can deal with it.

Take for example the extension of PicoJava with a substitution placeholder parameterized with the `Identifier` nonterminal. In the object language the identifier is used to link the type of the declaration to the type of the expression where the identifier is used. This link is based on the name of the identifier and when having a placeholder the same mechanism must be used. The substitution placeholder does not provide a name, but holds a metalanguage expression. Since the same expression used in different places reduces to the same value, it is possible to use the literal expression as identifier. This semantic property of the identifier placeholder is implemented via the code of Figure 8.9.

Adding placeholders to an object language can introduce syntactic ambiguities, see Section 6.7. For example, a Java template can contain an ambiguity where it is not clear whether a constructor declaration or method declaration is defined. Sometimes an ambiguity can be solved using syntactical typing, see Section 5.3.1. However, this is not always an option. When an ambiguity is found the parser instantiates a list node, containing the different sub parse trees. The SGLR parser instantiates these nodes on the fly, while parsing a template. In order to handle these ambiguities by JastAdd, we add for every nonterminal a special production rule to represent the ambiguity. We used a filter mechanism based on type-driven disambiguation to solve the ambiguities [114]. This mechanism checks for every ambiguity whether it has type errors or other semantic errors and if so, it removes that sub parse tree. When no correct subtree can be found, an error is generated. In case of templates it is always possible that multiple sub parse trees are correct. At that moment only a warning is generated, where the template developer can choose to solve the ambiguity manually by syntactical typing the placeholders.

8.5.3 Object Language Checks Dependent of Metalanguage

The last type of static semantic checks is where the metalanguage influences the object language. This occurs when a match-replace placeholder, a conditional placeholder or an iteration placeholder is applied. These placeholders do not insert a string from the input data tree into the code, but depending on values in the input data tree a piece of object code is enabled or disabled. The essence of these static semantic checks is that they should validate which combinations of object code are allowed. As it heavily depends on the input data whether the generated code will contain an error, these checks generate warnings instead of errors.

An example is a template with two not nested conditional placeholders, where the first one declares a variable and the second one uses that variable. The second conditional placeholder may only insert its code in the output code, when also the first conditional placeholder is enabled. Whether the first conditional placeholder inserts code in the output code is independent of the second conditional. A checker should generate a warning that the second conditional only may yield code when the first conditional placeholder yields its code. The static semantic checker for the input language should check that this combination is not possible.

In this section, we only consider the semantics of the conditional placeholder and iteration placeholders. The operational semantics of a conditional placeholder and iteration placeholder are simpler than the match-replace placeholder, which implies that the static semantic properties are also easier.

The conditional placeholder includes the `then` part or the `else` part into the code depending on the condition and the iteration includes zero or more time its body. For the conditional placeholder, it is checked that the `then` part or the `else` part are allowed. In case of the iteration placeholder we check whether it is allowed to include the body zero, one, or more times. For example, one can imagine that a variable declaration inside an iteration placeholder with a fixed identifier will lead to an error if the code is generated more than once. We check for this property, and return a warning when the identifier of the variable declaration is not a placeholder.

An example of such a check is to verify whether identifiers are a placeholder within an iteration placeholder. One can imagine that a variable declaration inside an iteration placeholder with a fixed identifier will lead to an error if the code is generated more than once. We check for this property, and return a warning when the identifier of the variable declaration is not a placeholder. A warning is generated since it is possible that some input data trees will not result in generated code with an error. For example when an iteration placeholder with a variable declaration with fixed identifier is generated one time. However, this is against the nature and goal of an iteration placeholder. Therefore we generate a warning. The check, shown in Figure 8.10, searches in the body of the iteration placeholder for variable declarations. If one is present, and the variable declared is not a placeholder, a warning is generated.

```
public void PhForeachBlockStmt.collectPhErrors(Collection c) {
  super.collectPhErrors(c);
  for (int i = 0; i < getNumBlockStmt(); i++) {
    if (getBlockStmt(i) instanceof VarDecl) {
      if (!(((VarDecl) getBlockStmt(i)).getIdentifier()
              instanceof PhIdentifier))
        warning(c, "Variable declarations in a Foreach" +
          "may only be about placeholders");
    }
  }
}
```

Figure 8.10 Checking function for the iteration placeholder of BlockStmt.

8.6 Related Work

Our template syntax is used as an add-on to an arbitrary object language. Our goal is to guarantee the syntax safety and semantic safety as much as possible, while being flexible with respect to the choice of the object language. Some languages support built-in template-like constructions. Examples are C++ templates or Java generics, in which classes can be parameterized by type information. The use of C++ templates is type safe, but this is only guaranteed at instantiation time. Another concept of built-in meta-programming is staged compilation, for example in a system like MetaML [110], where code can be executed in different execution stages. The type safety is ensured by extending the ML type checker with new scope rules for the staging levels. However, programs with stage annotations are complete and contrary to templates all information is available. C++ templates, Java generics or explicit staging are part of the language. Therefore, syntax checking and semantic checking is offered by the compiler. Constructions embedded in the language are only able to generate safe code for the language itself and the parameterizable nonterminals are fixed in the language specifications. While our aim is to be flexible with respect to the choice of the object language and to allow adding template constructs to every nonterminal in the object language.

Extending a static semantic checker to support multiple languages is presented in [26] for embedded domain specific languages (DSL) in some general purpose language (GPL). They connect the type systems of the DSL to the type system of the GPL by adding rules to the static semantic checker. This is possible since the DSL code is an abbreviation for GPL code and the additional static semantic rules reflect the mapping of the static semantic part. Compilation units contain DSL and GPL code, and they are complete, i.e. when the DSL is rewritten to GPL code it can be compiled and executed. In contrast to embedded DSLs, templates are incomplete code and some external information is necessary to complete them. Incomplete code implicates missing information and thus missing information for the static semantic checking.

Another safe template-like approach is SafeGen [61]. SafeGen is a tool to build Java generators. It uses an automatic theorem prover to prove the well-formedness of the generated code for all possible inputs of a generator. This approach heavily depends on the assumption that the input is a valid Java program and the knowledge of the Java type system. The template developer can define placeholders (cursors) to obtain data from the Java input program. Those placeholders must contain constraints based on their use in the template. A prover is used to check the constraints, which ensures that the template cannot generate ill-formed code. The approach of SafeGen is applicable for other input languages and object languages, but the implementations are not generic, since they depend heavily on the chosen (input and object) languages.

An approach for creating static semantic checkers for templates is discussed by Heidenreich et al. [57]. Instead of using grammars, they use (meta) models with OCL to express the structure and semantics of the object and metalanguage. They can extend an object language model with template constructs in a modular way. Their approach does not provide a solution to handle ambiguities, which is a requirement when handling templates. Furthermore, they require that the input data contains subtrees with the (syntactical) type of the object language where it is injected, while we allow a weak link between the external input data and object language.

8.7 Future Work

We discussed the implementation of a static semantic checker for PicoJava templates using JastAdd. Further research is necessary to scale up the approach to Java templates using JastAddJ and to improve re-usability of static semantic checkers. These topics are discussed in the following sections.

8.7.1 Checking Java Templates using JastAddJ

We implemented a static semantic checker for PicoJava templates. Since PicoJava is a small language for demonstration purposes, we want to scale up our approach and implement a static semantic checker for Java templates. Doing this with JastAdd seems a feasible option, since *JastAddJ*, a Java 1.4 compiler including a static semantic checker using JastAdd, is available [37]. Furthermore a complete SDF grammar exists for Java 1.4.

Adding template constructs to Java and re-using the static semantic checker JastAddJ seems obvious, but due to the construction of JastAddJ it was not possible to obtain a static semantic checker for Java templates based on JastAddJ. The Java SDF grammar and JastAddJ are not compatible. The differences can be classified in two categories:

◇ rewriting during parsing;

◇ instantiation of the Static Semantic Checker.

The abstract syntax definition of JastAddJ does not fully comply with the Java Language Specification [49]. On several places production rules are combined. The reason can be found in the fact that it is easy to pass attributes up and down through the tree in JastAdd, but not between sibling nodes. For example, the `method_header` and the `method_body` in Java are not hierarchically defined in the Java Language Specification, which makes it difficult in the `method_body` to obtain a list of formal parameters provided by the `method_header`. In JastAddJ the parser already combines these two nodes in a single node, called `method_declaration`.

Furthermore, during parsing the static semantic checker is called to obtain context related properties of nodes. For example, while parsing JastAddJ uses the static semantic checker to resolve the full qualified names of types.

In our approach the separation between the parser and the static semantic checker is absolute. The interference of the parser and JastAddJ is not a problem for connecting SGLR and JastAddJ in order to check Java code, but it has it effects on mapping the parse tree to JastAddJ. In the ideal situation the constructor names can directly be mapped to the JastAddJ classes, just as we did in the PicoJava implementation. This is not possible and the mapping must handle the rewriting and use of the static semantic checker to complete fields [95].

The previously mentioned grammar differences and transformations on the tree during the traversal phase make JastAddJ unsuitable to be extended with placeholder semantics. For the introduction of placeholders, the grammar definitions of the concrete syntax and the abstract syntax need to facilitate a one-to-one transformation of the parse tree. A transformation during the mapping will result in mappings on the occurrence of the placeholders. Since the location of the placeholders in the parse tree is important for the semantic properties of a placeholder, any transformation will result in additional dependencies on the placeholders. This does not imply that extending the Java static semantic checker of JastAddJ with placeholders is impossible. This merely indicates that design decisions have been made in JastAddJ that are not beneficial for extension it with rules for placeholders, with as key issue the lack of strict separation between parser and static semantic checker.

8.7.2 Re-usability

The re-usability of existing static semantic checkers is also a concern. Implementing a static semantic checkers from scratch is a tough and a time consuming task, which should be avoided whenever possible. We showed that the semantics of an object language can be extended in a modular fashion when new language constructs are added. However, we are not convinced that a generic approach is possible, which extends every static semantic checker of a given object language with placeholder constructs. Strict separation between the parser and static semantic checker is necessary to support re-use. When this requirement is not met, it is hard to re-use the static semantic checker and either a lot of work must be redone or a separate mapping code must be written. The question is, whether it is possible to define static semantic checkers in such way that they are re-usable. What is the right formalism to implement static semantic checkers? If re-using an existing static semantic checker is not an option, then it could be feasible to implement a static semantic checker for commonly used object languages, like XHTML and Java.

8.8 Conclusions

This chapter discussed our approach to implement static semantic checkers for templates. We discussed the implementation of a static semantic checker for PicoJava templates. The presented checks are sound, as a detected error will result in an error in the generated code or during template evaluation, but not complete. There is still a chance that the generated code contains static semantic errors and thus the template contains errors. In the first place, the level of completeness of checking a template depends on the object language checker. Subsequently, the amount of effort put in the specification of the placeholder semantics added to the original checker influences the level of checking. Finally, the more placeholders a template contains, the less a static semantic checker for templates can be used to guarantee that the output code is correct with respect to static semantics.

The use of the attribute grammar system JastAdd allows us to extend the original PicoJava static semantic checker without a lot of modifications. A kind of modularity is observed, where the static semantic checks for the metalanguage are independent of the original static semantic checker. Although, the requirements are stricter to re-use a static semantic checker than to re-use a context-free grammar. The static semantic checker must be strictly separated from the parser.

9 Conclusions

We studied the application of templates in the context of code generation in order to improve the quality of templates and to improve the quality of the code they instantiate. This chapter summarizes the contributions of this thesis. We provide suggestions for future work and we end with the final conclusions providing answers on the research questions from Chapter 1.

9.1 Contributions

Chapter 1 introduced the context of this thesis; the application of templates in the context of code generation. Code generators can be used to transform specifications at a higher level of abstraction to implementations at a lower level of abstraction. The use of specifications at a higher level of abstraction improves the manageability of the code and/or design of a program [46].

Code generators can be implemented using different approaches and techniques, including templates. A template is a text that contains placeholders, which are replaced to obtain an output text. Contemporary template evaluators are text-based, not offering protection against syntax errors in the template and the code they instantiate.

The central research question is *how the quality of template based code generators can be improved*. Quality, in general, is a broad notion and our scope is limited to the technical quality of templates and generated code. We focused on improving the maintainability of template based code generators and the correctness of the generated code. This is facilitated by the three main contributions provided by this thesis. First, the maintainability of template based code generators is increased by specifying the following requirement for our metalanguage. A too limited metalanguage can only be used in a limited number of cases. A too powerful metalanguage increases the risk of writing complete computer programs in a template, breaking the strict separation of concerns between model and view [92]. We used the theory of formal languages to specify our metalanguage. This contribution is discussed in Sections 9.1.1 and 9.1.2. Second, we ensure correctness of the templates and generated code. We have improved the (syntactical) correctness of the templates and the output code, discussed in Sections 9.1.3 and 9.1.4. Protection against syntax errors in the template provides a shorter development cycle, as code does not need to be generated to detect syntax errors in the object code. We have also presented static semantic checking of templates, discussed in Section 9.1.6. Third, the presented theory and techniques are validated by case studies, discussed in Section 9.1.5. These case studies show application of templates in real world applications, increased maintainability and syntactical correctness of generated code. The next sections discuss the contributions of this thesis.

9.1.1 Unparser Completeness

Our goal was to define a metalanguage that does not provide more computational power than necessary, without being too limited for code generation purposes. Chapter 3 discussed the requirements of a metalanguage for code generators to accomplish this goal. The requirements are based on the relations between concrete syntax, abstract syntax trees and their grammars. The mapping of abstract syntax to concrete syntax is called an unparser. The unparser should have two specific properties: parsing and desugaring its output results in the original abstract syntax tree of the used input, and the unparser can instantiate all meaningful sentences of the output language. The second property requires that an unparser has no limitation on the sentences it can instantiate. A metalanguage capable to express an unparser is powerful enough to express code generators.

The main contribution of this chapter is that a linear deterministic top-down tree-to-string transducer fulfills the requirements to implement an unparser. A metalanguage for implementing unparsers should at least be powerful enough to express a linear deterministic top-down tree-to-string transducer, otherwise some sentences of the output language cannot be instantiated. We call the ability of a metalanguage to implement an unparser *unparser-complete*. In case a metalanguage is beyond unparser-completeness, for example Turing complete, it increases the risk of writing complete programs in a template.

9.1.2 Our Metalanguage

We defined our metalanguage in Chapter 4. This metalanguage provides the constructs necessary to be unparser-complete. We provide two kernel constructs in our metalanguage: subtemplates and match-replace placeholders. We also introduced three derived constructs: substitution placeholders, iteration placeholders and conditional placeholders. These are abbreviations for combinations of subtemplates and match-replace placeholders.

In order to prevent writing model transformations in templates, our metalanguage cannot change the input data and it does not support complex expressions. This enforces a clear separation of model and view.

We compared our metalanguage with the metalanguages of the related template systems ERb, Velocity, JSP and StringTemplate. The selection of the template systems is based on the different metalanguages and availability of a working template evaluator. ERb, JSP and Velocity offer a Turing complete metalanguage, while StringTemplate only supports basic functionality, like subtemplates, substitution, iteration and conditions. We implemented an unparser for the PICO language in every system to compare their metalanguages on expressiveness. PICO is a toy language supporting most features of a context-free grammar. We looked at the limitations of the input data processing capabilities of the metalanguage and the limitations to instantiate the output code.

The StringTemplate implementation of the PICO unparser has the fewest lines of code, but in contrast with our metalanguage, StringTemplate cannot directly accept all regular trees. It can only handle unordered trees. An extra transformation is necessary to convert the input data from an ordered tree to an unordered tree. This conversion uses ordered lists to represent the unordered children of a node, where the StringTemplate meta code can fetch an indexed element using a number of list operations.

ERb, Velocity and JSP come with a Turing complete metalanguage. However, they do not have a block scoping mechanism for the meta-variables. A workaround for proper handling of meta-variable scopes was necessary to implement the PICO unparser. The consequence of this workaround was additional boilerplate code. Furthermore, rich metalanguages increase the chance of undesired programming in templates, which can result in tangling of concerns. For example, it is undesired to specify model transformations inside a template.

Our metalanguage can handle all input data trees defined by top-down deterministic tree automata and it can instantiate all context-free languages, while separation of concerns is enforced as model transformations cannot be expressed.

9.1.3 Syntactical Correctness of Templates

In Chapter 5 the topic is the syntactical correctness of templates. We have presented the construction of grammars containing production rules for the object language and metalanguage in a template. Syntax errors in the object code and meta code of a template are detected while parsing the template instead of dealing with syntax errors at compile time of the generated code. The complete template is parsed, and thus checked for syntax errors. The syntax of the templates is not different from text templates, as a result they provide the same user experience.

The template grammar is obtained by combining the object language grammar and placeholder grammar by adding the placeholder syntax as alternative to the object language nonterminals. The construction of such a template grammar is generic. Only a combination grammar connecting both languages has to be defined manually. Since we use SDF, this combination grammar uses *module parameters* to specialize the placeholder syntax for a specific nonterminal, instead of redefining the placeholder syntax for every nonterminal. The advantage of this approach is the ease of using off-the-shelf object language grammars [70].

9.1.4 Syntax Safe Evaluation

Parsing templates on its own is not sufficient to guarantee that the output of the template evaluator is a sentence of the output language. In Chapter 6 we presented syntax safe evaluation. Syntax safe evaluation provides a mechanism to detect syntax errors during the generation of the code. It prevents that placeholders in a template are being replaced by syntactical incorrect constructs. This guarantee is achieved by checking that the root nonterminal of the sub parse tree replacing a placeholder is equal to the object language nonterminal where the placeholder is applied.

The template evaluator is independent of the object language and does not need to be changed when another object language is used. It is even possible to use object code containing multiple languages, like Java with embedded SQL. We have implemented these ideas in a tool called *Repleo*.

9.1.5 Validation of Practical Applicability

In order to validate the real world applicability of syntax safe templates, Chapter 7 presented four case studies using templates. The case studies were chosen to show that our metalanguage in combination with the two-stage architecture results in better maintainable code and to show the benefits of syntax safety in different situations. The two-stage architecture exists of an explicit model transformation stage and code emitter stage.

The first case study covered the generation of web application back-ends. The generated code is based on a three tier MVC architecture, the code generator must instantiate code for the different layers expressed in different languages from a single input model. The second case study was the reimplementation of ApiGen. ApiGen is an application to generate a Java API for creating, manipulating and querying tree-like data structures represented as ATerms. It covers the generation of Java code based on the Factory pattern and Composite pattern. The third case study was the reimplementation of NunniFSMGen. NunniFSMGen is a tool to generate finite state machines from a transition table. It covers the generation of behavioral code for different output languages. The generated state machine is based on the state design pattern. The final case study showed that syntax safety can improve the safety of dynamic code generation in web applications. It covers code generation during the usage of an application, where syntax safety is used to reduce the possibility of security breaches.

For the first three case studies, the use of a two-stage architecture results in a better separation of concerns between the model transformation and code emitters. In case of ApiGen and NunniFSMGen it was possible to compare the original implementation with the reimplementation. The last case study shows the benefits of syntax safety when generating XHTML in a dynamic web application.

The compact metalanguage, as designed in Chapter 4, enforced us to use the two-stage architecture. It is not possible to express all model transformations in our metalanguage, as calculations cannot be expressed and intermediate values cannot be stored. The metalanguage is unparser-complete; it is not a limitation of the code the templates can instantiate. This is empirically validated by the four presented case studies, since they cover a broad range of code generation application areas.

The benefits of the two-stage architecture are reflected in the reduced number of lines of code. The reimplemented code generators are between two and three times as small as the original implementations. The NunniFSMGen case study shows that the choice of the output language is made at the view level, i.e. the templates, while the input data for the templates is not changed.

The last case study shows that syntax safety increases the safety of dynamic code generation in web applications. Cross-site scripting is prevented without introducing a lot of boilerplate code and checking code. Grammars are used to specify allowed (sub) sentences in the XHTML template instead of implementing the checks for the input data in the controller component of a web application.

Syntax safe templates as presented in this thesis can be used in the different application areas of the discussed case studies without any adaptation. This shows that syntax safe templates are applicable in real world code generator cases.

9.1.6 Towards Static Semantic Checking

We also showed that is possible to check static semantic properties of templates. Static semantic checks are defined for the metalanguage, for the object language and checks for the situations where the object language is dependent on the metalanguage.

We implemented a prototype of a static semantic checker for PicoJava templates using the attribute grammar system JastAdd. The main reason for using JastAdd is that it claims to offer modular specifications of compiler tools [38]. We specified a prototype of a static semantic checker for PicoJava templates based on JastAdd. PicoJava is used as object language since JastAdd is distributed with a static semantic checker for it.

The presented checks are sound, as a detected error will result in an error in the generated code or during template evaluation, but not complete. There is still a chance that the generated code contains static semantic errors and thus the template contains errors. In the first place, the level of completeness of checking a template depends on the object language checker. Subsequently, the amount of effort put in the specification of the placeholder semantics added to the original checker influences the level of checking.

The use of the attribute grammar system JastAdd allows us to extend the original PicoJava static semantic checker without a significant number of modifications. A kind of modularity is observed, where the static semantic checks for the metalanguage are independent of the original static semantic checker. In that case, it is a requirement that the static semantic checker is strictly separated from the parser.

9.2 Future Work

We illuminated a number of issues playing a role in the use of templates for code generators. This section presents directions of future work in this context.

First, we did not study the performance of the Repleo implementation. We used an average contemporary desktop computer[1] to generate the code in the case studies. The code was generated between a few seconds and a couple of minutes depending on the size of the templates, complexity of the templates and size of the input data. When generating a large amount of source code, the execution time is not a bottleneck, but the current implementation will not scale when syntax safe evaluation is used in dynamic code generation systems. Future work is to investigate optimization of the syntax safe evaluator in order to be usable in industrial dynamic code generation, like web applications. Our hypothesis is that using partial evaluation [44] to reduce the amount of work to be done at runtime results in fast template evaluation, without losing the syntax safe properties.

Second, our aim was to increase the level of fidelity of using templates, without being experienced as more complex than text templates. Although we did not

[1] Intel® Core^{TM} 2 CPU 2.40 GHz with 2 GB RAM.

experience an increased difficulty, the research question is still open, whether our syntax safe templates and metalanguage are indeed as complex as the metalanguages offered by text templates.

The last topic of future work is to to extend the work of static semantic checkers for templates. Chapter 8 discussed an implementation of static semantic checker for PicoJava templates. The future work related to static semantic checkers is divided in scaling up the approach to Java templates using JastAddJ and improving re-usability of static semantic checkers.

First, adding template constructs to Java and re-using an existing implementation of static semantic checker of Java called JastAddJ seems obvious, but due to the construction of JastAddJ it was not possible to obtain a static semantic checker for Java templates based on JastAddJ. The Java SDF grammar and JastAddJ are not compatible. Rewriting during parsing and instantiation of the static semantic checker while parsing are the causes of this incompatibility.

Second, the re-usability of existing static semantic checkers is also a concern. We showed that the semantics of an object language can be extended in a modular fashion when new language constructs are added. However, we are not convinced that a generic approach is possible, which extends every static semantic checker of a given object language with placeholder constructs. The question is, whether in the general case it is possible to define static semantic checkers in such way that they are re-usable.

9.3 Final Conclusions

Recurring on the central research question, whether the quality of template based code generators can be improved. This thesis showed that the quality of template based code generators can be increased comparing to text-based templates using the theory of formal languages. The central research question is refined in six more specific research questions, see Chapter 1.

The topic of the first and second research question is the metalanguage. The metalanguage does not need more computational power than a linear deterministic top-down tree-to-string transducer offers to be able to instantiate context-free languages. In the context of a two-stage architecture, this limited computational power enforces a strict separation of concerns between the model transformer and the code emitters.

The syntactical correctness of the templates and generated code is the topic of the third and fourth research question. The use of grammars for templates, including production rules for the object language, prevents syntax errors in the template and in the generated code. Debugging templates is better supported as the origin of the (syntax) error can be better determined. A side-effect of syntax safe template evaluation is out-of-the-box protection against code injection attacks.

The fifth research question deals with practical applicability. Four case studies showed that syntax safe templates are suitable in real world code generator cases.

The topic of the sixth research question is static semantic checking of templates. We discussed object language checks, metalanguage checks and checks for the situations where the object language is dependent on the metalanguage. We implemented a prototype of a static semantic checker for PicoJava templates using attribute grammars. The use of attribute grammars leads to re-use of the original PicoJava checker.

Bibliography

[1] International Standard: Information technology – Syntactic metalanguage – Extended BNF. Technical Report ISO/IEC 14977: 1996(E), December 1996. (Cited on page 205.)

[2] *Oxford Advanced Learner's Dictionary*. Oxford University Press, 7th edition, 2005. (Cited on pages 1, 2, and 7.)

[3] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989. (Cited on pages 33 and 148.)

[4] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, 1977. (Cited on page 148.)

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1986. (Cited on pages 28, 32, 130, 148, 174, 204, and 214.)

[6] M. Alpuente, M. Falaschi, M. J. Ramis, and G. Vidal. A compositional semantics for conditional term rewriting systems. In *ICCL '94: International Conference on Computer Languages*, pages 171–182, Piscataway, NJ, USA, 1994. IEEE Computer Society Press. (Cited on page 67.)

[7] L. Andersen. *JDBC(tm) 4.0 Specification*. Sun Microsystems, Inc., November 2006. (Cited on page 140.)

[8] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998. (Cited on page 15.)

[9] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, 1960. (Cited on page 38.)

[10] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the STL and some general implications. In *ICSE '05: International Conference on Software Engineering*, pages 451–459, New York, NY, USA, 2005. ACM Press. (Cited on page 6.)

[11] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2nd edition, 2003. (Cited on page 3.)

[12] H. Bergsten. *Javaserver Pages*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. (Cited on pages 86, 88, and 91.)

[13] J. A. Bergstra, J. Heering, and P. Klint. *Algebraic specification*. ACM Press, New York, NY, USA, 1989. (Cited on pages 15, 17, 37, and 132.)

[14] G. J. Bex, F. Neven, and J. van den Bussche. DTDs versus XML schema: a practical study. In *WebDB '04: International Workshop on the Web and Databases*, pages 79–84, New York, NY, USA, 2004. ACM Press. (Cited on page 158.)

[15] P. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *RWLW '96: First International Workshop on Rewriting Logic and its Applications*, volume 4, pages 35–50. Electronic Notes in Theoretical Computer Science, 1996. (Cited on page 15.)

[16] B. Bos and H. W. Lie. *Cascading Style Sheets, level 1*. W3C, 1996. http://www.w3.org/TR/CSS1/ (accessed on November 30, 2010). (Cited on page 189.)

[17] G. Bracha. Generics in the Java Programming Language. Technical report, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf (accessed on November 30, 2010). (Cited on page 11.)

[18] M. G. J. van den Brand. *PREGMATIC: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992. (Cited on page 204.)

[19] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12:152–190, April 2003. (Cited on page 171.)

[20] M. G. J. van den Brand, P. E. Moreau, and J. J. Vinju. A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings Software*, 152(2):70–78, April 2005. (Cited on pages 12 and 14.)

[21] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *CC '01: Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag. (Cited on pages 120 and 157.)

[22] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000. (Cited on pages 41, 73, 114, and 157.)

[23] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *CC '02: Compiler Construction*, pages 143–158, London, UK, 2002. Springer-Verlag. (Cited on page 140.)

[24] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996. (Cited on page 50.)

[25] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *GPCE '07: Generative Programming and Component Engineering*, pages 3–12, New York, NY, USA, 2007. ACM Press. (Cited on pages 131, 139, 143, and 199.)

[26] M. Bravenboer, R. de Groot, and E. Visser. MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT. In *GTTSE '05: Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 297–311, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on pages 143 and 218.)

[27] M. Bravenboer, É. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. In *OOPSLA '06: Object-Oriented Programming Systems, Languages, and Applications*, pages 209–228, New York, NY, USA, 2006. ACM Press. (Cited on page 118.)

[28] J. T. Chan and W. Yang. Ambiguities in Java. In *CTHPC '02: Workshop on Compiler Techniques for High-Performance Computing*, pages 51–62, Hualien, Taiwan, 2002. (Cited on page 205.)

[29] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS'03: International Conference on Static Analysis*, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag. (Cited on page 172.)

[30] J. Clark. W3C recommendation. XSL Transformations (XSLT) version 1.0, 1999. http://www.w3.org/TR/xslt (accessed on November 30, 2010). (Cited on page 15.)

[31] L. G. W. A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. PhD thesis, Technische Universiteit Eindhoven, 2008. (Cited on pages 27, 33, 34, 73, and 77.)

[32] L. G. W. A. Cleophas. Private communication, September 2009. (Cited on page 47.)

[33] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata (accessed

on November 30, 2010), 2008. release November, 18th 2008. (Cited on pages 27, 30, 31, 33, 34, 47, 55, and 73.)

[34] J. Conallen. Modeling Web application architectures with UML. *Communications of the ACM*, 42(10):63–70, 1999. (Cited on pages 20 and 21.)

[35] D. Crockford. RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON). IETF RFC, 2008. (Cited on pages 99, 197, and 198.)

[36] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Mélèse. Document structure and modularity in Mentor. *ACM SIGPLAN Notices*, 19(5):141–148, 1984. (Cited on page 36.)

[37] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. *ACM SIGPLAN Notices*, 42(10):1–18, 2007. (Cited on pages 205 and 219.)

[38] T. Ekman and G. Hedin. The JastAdd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007. (Cited on pages 205 and 228.)

[39] J. Engelfriet. Tree Automata and Tree Grammars. Manual written lecture notes, 1974. (Cited on pages 27, 28, 29, 30, and 47.)

[40] J. Engelfriet, G. Rozenberg, and G. Slutzki. Tree transducers, L systems, and two–way machines. *Journal of Computer and System Sciences*, 20(2):150–202, 1980. (Cited on page 55.)

[41] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002. (Cited on pages 103 and 119.)

[42] J. M. Favre. Towards a Basic Theory to Model Model Driven Engineering. In *WIME '04: Workshop on Software Model Engineering*, 2004. (Cited on page 7.)

[43] L. Floridi and J. W. Sanders. Levellism and the Method of Abstraction. Technical Report 22.11.04, Oxford University, 2004. (Cited on pages 3 and 5.)

[44] Y. Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler–Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. (Cited on pages 64 and 228.)

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1995. (Cited on pages 146, 157, 160, 163, 173, and 178.)

[46] A. Gerstlauer and D. D. Gajski. System-level abstraction semantics. In *ISSS '02: International Symposium on System Synthesis*, pages 231–236, New York, NY, USA, 2002. ACM Press. (Cited on page 223.)

[47] C. Goddard. *Semantic analysis: A practical introduction*. Oxford University Press, 1998. (Cited on page 7.)

[48] D. Goodman and B. Eich. *JavaScript Bible*. Wiley & Sons, New York, NY, USA, 4th edition, 2000. (Cited on page 189.)

[49] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2000. (Cited on pages 205 and 220.)

[50] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1977. (Cited on page 151.)

[51] J. Hartmanis. Context-free languages and Turing machine computations. In *Symposia in Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 42–51. Amer Mathematical Society, 1967. (Cited on page 31.)

[52] I. Hayes, editor. *Specification case studies*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1987. (Cited on pages 68 and 69.)

[53] G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3): 301–317, 2000. (Cited on page 206.)

[54] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47:37–58, April 2003. (Cited on pages 172, 205, 206, 207, and 211.)

[55] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF — reference manual. *ACM SIGPLAN Notices*, 24:43–75, November 1989. (Cited on pages 38, 40, 131, and 205.)

[56] J. Heering and M. Mernik. Domain-Specific Languages in Perspective. Technical Report SEN-E0702, Centrum voor Wiskunde en Informatica, September 2007. (Cited on pages 3 and 5.)

[57] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, and M. Böhme. Generating safe template languages. In *GPCE '09: Generative Programming and Component Engineering*, pages 99–108, New York, NY, USA, 2009. ACM Press. (Cited on pages 142 and 219.)

[58] J. Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, CT, USA, 2003. (Cited on pages 8, 21, 86, 88, and 154.)

[59] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2001. (Cited on page 27.)

[60] G. Hotz. Normal-form transformations of context-free grammars. *Acta Cybernetica*, 4:65–84, 1980. (Cited on page 31.)

[61] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *GPCE '05: Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326, Berlin, Heidelberg, 2005. Springer-Verlag. (Cited on pages 118 and 219.)

[62] J. Hunter. The problems with JSP. http://www.servlets.com/soapbox/problems-jsp.html (accessed on November 30, 2010), 2000. (Cited on page 99.)

[63] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. (Cited on page 193.)

[64] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC. (Cited on page 205.)

[65] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA, 1975. (Cited on page 45.)

[66] E. Kang and M. D. Aagaard. Improving the Usability of HOL Through Controlled Automation Tactics. In *TPHOLs '07: Theorem Proving in Higher Order Logics*, volume 4732, pages 157–172, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 15.)

[67] G. Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *IMSA '92: International Workshop on New Models for Software Architecture: Workshop on Reflection and Meta-Level Architecture*, pages 1–11, Tokyo, Japan, 1992. (Cited on page 4.)

[68] G. Kiczales and E. Hilsdale. Aspect-oriented programming. *SIGSOFT Software Engineering Notes*, 26:313, September 2001. (Cited on pages 207 and 209.)

[69] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, 30(5):187–196, 2005. (Cited on page 6.)

[70] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, 2005. (Cited on pages 104, 118, 119, and 226.)

[71] P. Klint, T. van der Storm, and J. J. Vinju. EASY Meta-Programming with RASCAL. In *GTTSE '09: Generative and Transformational Techniques in Software Engineering*, volume 6491 of *Lecture Notes in Computer Science*, pages 185–238, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 19.)

[72] P. Klint, T van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM '09: International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, Los Alamitos, CA, USA, 2009. IEEE Computer Society Press. (Cited on page 17.)

[73] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968. (Cited on pages 205 and 206.)

[74] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LFP '86: LISP and Functional Programming*, pages 151–161, New York, NY, USA, 1986. ACM Press. (Cited on page 79.)

[75] S. Kong, W. Choi, and K. Yi. Abstract parsing for two-staged languages with concatenation. In *GPCE '09: Generative Programming and Component Engineering*, pages 109–116, New York, NY, USA, 2009. ACM Press. (Cited on page 119.)

[76] J. W. C. Koorn. *Generating uniform user-interfaces for interactive programming environments*. PhD thesis, Universiteit van Amsterdam, 1994. (Cited on page 36.)

[77] A. Krall. Efficient JavaVM Just-in-Time Compilation. In *PACT '98: Parallel Architectures and Compilation Techniques*, pages 205–213, Washington, DC, USA, 1998. IEEE Computer Society Press. (Cited on page 189.)

[78] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988. (Cited on pages 148 and 149.)

[79] D. Leijen and E. Meijer. Domain specific embedded compilers. *ACM SIGPLAN Notices*, 35:109–122, December 1999. (Cited on pages 12 and 144.)

[80] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. (Cited on page 5.)

[81] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. *IEEE International Symposium on Software Metrics*, pages 87–94, 2002. (Cited on page 6.)

[82] L. Moonen. Generating Robust Parsers using Island Grammars. In *WCRE '01: Working Conference on Reverse Engineering*, pages 13–23, Washington, DC, USA, 2001. IEEE Computer Society Press. (Cited on page 118.)

[83] P. E. Moreau, C. Ringeissen, and M. Vittek. A Pattern-Matching Compiler. In *LDTA '01: Language Descriptions, Tools and Applications*, volume 44, pages 161–180. Electronic Notes in Theoretical Computer Science, 2001. (Cited on page 213.)

[84] P. E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *CC '03: Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag. (Cited on page 77.)

[85] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195 – 228, 2004. (Cited on page 205.)

[86] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005. (Cited on page 158.)

[87] H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. Wiley & Sons, New York, NY, USA, 1992. (Cited on page 36.)

[88] A. Nijholt. The CYK approach to serial and parallel parsing. *Language Research*, 27(2):229–254, June 1991. (Cited on page 130.)

[89] G. van Emde Boas. Template Programming for Model-Driven Code Generation. In *19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, CA*, 2004. (Cited on page 22.)

[90] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35–61, 2004. (Cited on pages 145, 157, and 158.)

[91] N. S. Papaspyrou. *A formal semantics for the C programming language*. PhD thesis, National Technical University of Athens, Department of Electrical and Computer Engineering, 1998. (Cited on pages 204 and 205.)

[92] T. J. Parr. Enforcing Strict Model-View Separation in Template Engines. In *WWW '04: International Conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM Press. (Cited on pages 20, 21, 61, 86, 88, 98, 200, and 224.)

[93] T. J. Parr and R. W. Quong. ANTLR: a predicated-LL(k) parser generator. *Software: Practice & Experience*, 25(7):789–810, 1995. (Cited on page 45.)

[94] T.J. Parr. ANTLR Parser Generator. http://www.antlr.org (accessed on November 30, 2010). (Cited on page 33.)

[95] A. J. Peeters. Introducing Static Semantic Analysis to Templates Using JastAdd. Master's thesis, Technische Universiteit Eindhoven, 2009. http://alexandria.tue.nl/extra2/afstversl/wsk-i/peeters2009.pdf (accessed on November 30, 2010). (Cited on pages 212 and 220.)

[96] S. Pemberton, D. Austin, J. Axelsson, T. Çelik, D. Dominiak, H. Elenbaas, B. Epperson, M. Ishikawa, S. Matsui, S. McCarron, A. Navarro, S. Peruvemba, R. Relyea, S. Schnitzenbaumer, and P. Stark. XHTML 1.0 The Extensible Hypertext Markup Language. W3C Recommendation, August 2002. (Cited on pages 189 and 193.)

[97] N. Ramsey. Unparsing expressions with prefix and postfix operators. *Software: Practice & Experience*, 28(12):1327–1356, 1998. (Cited on page 50.)

[98] N. Ramsey. Pragmatic aspects of reusable program generators. *Journal of Functional Programming*, 13(3):601–646, 2003. (Cited on page 8.)

[99] M. R. Rieback, B. Crispo, and A. S. Tanenbaum. Is Your Cat Infected with a Computer Virus? In *PERCOM '06: International Conference on Pervasive Computing and Communications*, pages 169–179, Washington, DC, USA, 2006. IEEE Computer Society Press. (Cited on page 191.)

[100] M. Roth and E. Pelegrí-Llopart. JavaServer Pages Specification Version 2.0, 2003. Sun Microsystems, Inc. (Cited on pages 86, 88, and 91.)

[101] P. P. Rubens. Hendrik IV in de slag bij Ivry, 1628–1630. ©Rubenshuis, Antwerpen, Belgium. (Cited on page 2.)

[102] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006. (Cited on page 7.)

[103] T. Sheard. Accomplishments and Research Challenges in Metaprogramming. In *SAIG 2001: International Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44, London, UK, 2001. Springer-Verlag. (Cited on pages 6, 7, 8, 9, and 104.)

[104] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, pages 1–16, New York, NY, USA, October 2002. ACM Press. (Cited on page 11.)

[105] G. J. W. M. Smeets. A Domain Specific Language for Web Information Systems. Master's thesis, Technische Universiteit Eindhoven, 2010. http://alexandria.tue.nl/extra1/afstversl/wsk-i/smeets2010.pdf (accessed on November 30, 2010). (Cited on page 151.)

[106] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001. (Cited on page 173.)

[107] J. Spolsky. The law of leaky abstractions. http://www.joelonsoftware.com/articles/LeakyAbstractions.html (accessed on November 30, 2010), 2002. Joel on Software. (Cited on page 4.)

[108] T. Sturm, J. von Voss, and M. Boger. Generating Code from UML with Velocity Templates. In *UML '02: International Conference on The Unified Modeling Language*, pages 150–161, London, UK, 2002. Springer-Verlag. (Cited on page 15.)

[109] Sun Microsystems, Inc. *JDBC API*. http://java.sun.com/javase/technologies/database/ (accessed on November 30, 2010). (Cited on page 140.)

[110] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000. (Cited on pages 6, 10, 204, and 218.)

[111] L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems*, 30(6):1–40, 2008. (Cited on pages 11 and 12.)

[112] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2003. (Cited on page 11.)

[113] T. L. Veldhuizen. C++ Templates as Partial Evaluation. In *PEPM '99: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, New York, NY, USA, 1999. ACM Press. (Cited on page 11.)

[114] J. J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, Universiteit van Amsterdam, 2005. (Cited on pages 130 and 216.)

[115] J. J. Vinju. Type-Driven Automatic Quotation of Concrete Object Code in Meta Programs. In *RISE '05: Rapid Integration of Software Engineering Techniques*, volume 3943 of *Lecture Notes in Computer Science*, pages 97–112, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on page 111.)

[116] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In *GTTSE '07: Generative and Transformational Techniques in Software Engineering*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Berlin, Heidelberg. Springer-Verlag. (Cited on page 15.)

[117] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. http://www.science.uva.nl/pub/programming-research/reports/1997/P9707.ps.Z (accessed on November 30, 2010). (Cited on pages 38, 45, 130, and 210.)

[118] E. Visser. Stratego: A language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *RTA '01: Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361, Berlin, Heidelberg, 2001. Springer-Verlag. (Cited on page 15.)

[119] E. Visser. Meta-Programming with Concrete Object Syntax. In *GPCE '02: Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Berlin, Heidelberg, October 2002. Springer-Verlag. (Cited on pages 17, 117, and 124.)

[120] M. Völter. A collection of patterns for program generation. In *EuroPLoP '03: European Conference on Pattern Languages of Programs*, 2003. (Cited on pages 7 and 12.)

[121] M. Völter and A. Gärtner. Jenerator – Generative Programming for Java. In *OOPSLA '01: Workshop on Generative Programming*, 2001. (Cited on pages 12 and 13.)

[122] G. Wachsmuth. A Formal Way from Text to Code Templates. In *FASE '09: Fundamental Approaches to Software Engineering*, pages 109–123, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on pages 142 and 143.)

[123] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08: International Conference on Software Engineering*, pages 171–180, New York, NY, USA, 2008. ACM Press. (Cited on pages 189 and 192.)

[124] D. S. Wile. Abstract syntax from concrete syntax. In *ICSE '97: International Conference on Software engineering*, pages 472–480, New York, NY, USA, 1997. ACM Press. (Cited on page 45.)

Summary

An Illumination of the Template Enigma

Software Code Generation with Templates

Creating software is a process of refining a concept to an implementation. This process consists of several stages represented by documents, models and plans at several levels of abstraction. Mostly, the refinement process requires creativity of the programmers, but sometimes the task is boring and repetitive.

This repetitive work is an indication that the program is not written at the most suitable level of abstraction. The level of abstraction offered by the used programming language might be too low to remove the recurring code. Code generators can be used to raise the level of abstraction of program specifications and to automate the repetitive work. This thesis focuses on code generators based on templates.

Templates are one of the techniques to implement a code generator. Templates allow extension of the syntax of a programming language, enabling generative programming without modifying the underlying compiler. Four artifacts are involved in a template based generator: templates, input data, a template evaluator and output code. The templates we consider are a concrete (incomplete) representation of the output document, i.e. object code, that contains holes, i.e. the meta code. These holes are filled by the template evaluator using information from the input data to obtain the output code. Templates are widely used to generate HTML code in web applications. They can be used for generating all kinds of text, like e-mails or (source) code. In this thesis we limit the scope to the generation of source code.

The central research question is *how the quality of template based code generators can be improved*. Quality, in general, is a broad notion and our scope is limited to the technical quality of templates and generated code. We focused on improving the maintainability of template based code generators and the correctness of the generated code. This is facilitated by the three main contributions provided by this thesis. First, the maintainability of template based code generators is increased by specifying the following requirement for our metalanguage. Our metalanguage should not be rich enough to allow programming in templates, without being too restrictive to express some code generators. We used the theory of formal languages to specify our metalanguage. Second, we ensure correctness of the templates and generated code. Third, the presented theory and techniques are validated by case studies. These case studies show

application of templates in real world applications, increased maintainability and syntactical correctness of generated code.

Our metalanguage should not be rich enough to allow programming in templates, without being too restrictive to express some code generators. The theory of formal languages is used to specify the requirements for our metalanguage. As we only consider to generate programming languages, it is sufficient to support the generation of languages defined by context-free grammars. This assumption is used to derive a metalanguage, that is rich enough to specify code generators that are able to instantiate all possible sentences of a context-free language. A specific case of a code generator, the *unparser*, is a program that can instantiate all sentences of a context-free language. We proved that an unparser can be implemented using a linear deterministic top-down tree-to-string transducer. We call this property *unparser*-completeness. Our metalanguage is based on a linear deterministic top-down tree-to-string transducer.

Recall that the goal of specifying the requirements of the metalanguage is to increase the maintainability of template based code generators, without being too restrictive. To validate that our metalanguage is not too restrictive and leads to better maintainable templates, we compared it with four off-the-shelf text template systems by implementing an unparser. We have observed that the industrial template evaluators provide a Turing complete metalanguage, but they do not contain a block scoping mechanism for the meta-variables. This results in undesired additional boilerplate meta code in their templates.

The second contribution is guaranteeing the correctness of the generated code. Correctness of the generated code can be divided in two concerns: syntactical correctness and semantical correctness. We start with syntactical correctness of the generated code. The use of text templates implies that syntactical correctness of the generated code can only be detected at compilation time. This means that errors detected during the compilation are reported on the level of the generated code. The developer is required to trace back manually the errors to their origin in the template or input data.

We believe that programs manipulating source code should not consider the object code as text to detect errors as early as possible. We present an approach where the grammars of the object language and metalanguage can be combined in a modular way. Combining both grammars allows parsing both languages simultaneously. Syntax errors in both languages of the template will be found while parsing it.

Moreover, only parsing a template is not sufficient to ensure that the generated code will be free of syntax errors. The template evaluator must be equipped with a mechanism to guarantee its output will be syntactically correct. We discuss our mechanism in short. A parse tree is constructed during the parsing of the template. This tree contains subtrees for the object code and subtrees for the meta code. While evaluating the template, subtrees of the meta code are substituted by object code subtrees. The template evaluator checks whether the root nonterminal of the object code subtree is equal to the root nonterminal of the meta code subtree. When both are equal, it is allowed to substitute the meta code. When the root nonterminals are distinct an accurate error message is generated. The template evaluator terminates when all meta code subtrees are substituted. The result is a parse tree of the object language and thus syntactically correct. We call this process *syntax safe* code generation.

In order to validate that the presented techniques increase maintainability and ensure syntactical correctness, we implemented our ideas in a syntax safe template evaluator called *Repleo*. Repleo has been applied in four case studies. The first case is a real world situation, where it is required to generate a three tier web application from a data model. This case showed that multiple layers of an applications defined in different programming languages can be generated from a single model. The second case and third case are used to show that our metalanguage results in a better maintainable code generator. Our metalanguage forces to use a two layer code generator with separation of concerns between the two layers, where the original implementations are less modular. The last case study shows that ensuring syntactical correctness results in the prevention of cross-site scripting attacks in dynamic generation of web pages.

Recall that one of our goals was ensuring the correctness of the generated code. We also showed that is possible to check static semantic properties of templates. Static semantic checks are defined for the metalanguage, for the object language and checks for the situations where the object language is dependent on the metalanguage. We implemented a prototype of a static semantic checker for PicoJava templates using attribute grammars. The use of attribute grammars leads to re-use of the original PicoJava checker.

Summarizing, in this thesis we have formulated the requirements for a metalanguage and discussed how to implement a syntax safe template evaluator. This results in better maintainable template based code generators and more reliable generated code.

Samenvatting

An Illumination of the Template Enigma

Software Codegeneratie met Sjablonen

Het creëren van software is een proces waarin een concept steeds verder wordt verfijnd tot een implementatie. Dit proces bestaat uit meerdere stappen bestaande uit documenten, modellen en ontwerpen op verschillende abstractieniveaus. Het verfijnen van een concept naar een implementatie vergt vaak creativiteit van de programmeurs, maar kan ook saai en herhalend werk zijn.

Het herhalende werk is een indicatie dat een programma niet wordt beschreven op het juiste abstractieniveau. Het abstractieniveau van de gebruikte programmeertaal kan te laag zijn om de herhalende code te vermijden. Codegeneratoren kunnen gebruikt worden om het abstractieniveau van programmaspecificaties te verhogen en het herhalende werk daarmee te automatiseren. Dit proefschrift richt zich op het gebruik van codegeneratoren op basis van sjablonen.

Het gebruik van sjablonen is een van de technieken om een codegenerator te implementeren. Een sjabloon breidt een programmeertaal uit om het geschikt te maken voor codegeneratie zonder de oorspronkelijke *compiler* (vertaler) aan te passen. Vier artefacten spelen een rol in een sjabloongebaseerde codegenerator: sjablonen, invoerdata, een sjabloonevaluator en de uitvoercode. De sjablonen, zoals wij ze beschouwen, zijn concrete (onvolledige) representaties van het uitvoerdocument, de zogenaamde objectcode, waarin zich gaten bevinden, de zogenaamde metacode. Deze gaten worden gevuld door de sjabloonevaluator met de informatie uit de invoerdata. Het resultaat van de sjabloonevaluator is de uitvoercode. Sjablonen worden al veelvuldig toegepast om HTML te genereren in webapplicaties en kunnen worden toegepast voor het genereren van allerlei soorten tekst, zoals e-mails of broncode. Wij beperken ons echter in dit proefschrift tot het genereren van broncode.

De centrale onderzoeksvraag is *hoe de kwaliteit van sjabloongebaseerde codegeneratoren kan worden verbeterd*. Kwaliteit is een algemeen begrip en we beperking ons tot de technische kwaliteit van sjablonen en gegenereerde code. We richten ons op de onderhoudbaarheid van sjablonen en de correctheid van de gegenereerde code. Dit is gefaciliteerd door de drie bijdragen van dit proefschrift: Ten eerste om de onderhoudbaarheid van sjablonen te verbeteren hebben we de volgende eis gesteld aan onze metataal. Onze metataal zou niet rijk genoeg moeten zijn om programmeren toe te staan, zonder dat de metataal een beperkende factor is om bepaalde codegeneratoren uit te drukken. We hebben formele taaltheorie toegepast om onze metataal te specificeren. Ten tweede garanderen we de correctheid van sjablonen en gegenereerde code. Als laatste hebben we met behulp van casussen aangetoond dat de gepresenteerde technieken industrieel toepasbaar zijn, de onderhoudbaarheid verbeteren en de syntactische correctheid van de gegenereerde code garanderen.

Een metataal zou niet rijk genoeg moeten zijn om programmeren toe te staan, zonder dat het een beperkende factor is om bepaalde codegeneratoren uit te drukken. We hebben formele taaltheorie gebruikt om de eisen voor de metataal te bepalen. Aangezien we ons beperken tot programmeertalen, is het voldoende om talen te ondersteunen die worden beschreven door een contextvrije grammatica. Deze aanname is gebruikt om de metataal af te leiden. Deze metataal is rijk genoeg om codegeneratoren te specificeren die in staat zijn alle mogelijke zinnen van een contextvrije taal te instantiëren. Een specifiek geval van een codegenerator, een *unparser* (de-ontleder), is een programma dat alle mogelijke zinnen van een contextvrije taal kan produceren. We hebben aangetoond dat een top-down lineaire deterministische boom–naar–tekenreeksomzetter krachtig genoeg is om een unparser uit te drukken. We noemen zijn eigenschap *unparservolledig* (de-ontledervolledig). Onze metataal is gebaseerd op deze boom–naar–tekenreeksomzetter.

Het doel van de beperkte metataal is om de onderhoudbaarheid van sjablonen te vergroten, zonder dat de taal een beperking oplegt. Om te valideren dat onze metataal krachtig genoeg is en de onderhoudbaarheid is verbeterd hebben we onze metataal vergeleken met de metataal van vier bestaande tekstgebaseerde sjabloonsystemen door middel van het implementeren van een unparser. Hoewel de industriële aanpakken waren voorzien van een Turingvolledige metataal zijn ze echter niet uitgerust met een geschikt scopingmechanisme voor de metavariabelen. Dit resulteert in ongewenste additionele metacode in hun sjablonen.

De tweede bijdrage is het garanderen van de correctheid van sjablonen en gegenereerde code. Correctheid kan worden onderverdeeld in twee zaken: syntactische correctheid en statisch semantische correctheid. We beginnen met discussie van het garanderen van de syntactische correctheid. Het gebruik van tekstgebaseerde sjablonen impliceert dat de syntactische correctheid van de gegenereerde code pas gedetecteerd kan worden tijdens de compilatie ervan. Dit betekent dat de foutmeldingen verwijzen naar de gegenereerde code. De ontwikkelaar zal handmatig de oorsprong van de fout moeten achterhalen in het sjabloon of in de invoerdata.

Om fouten eerder te ontdekken moet het voorkomen worden dat programma's die broncode manipuleren de objectcode beschouwen als tekst. We hebben een aanpak gepresenteerd welke de grammatica van de objecttaal en de metataal modulair combineert. Het resultaat is een grammatica die in staat is beide talen in een sjabloon tegelijk te ontleden. Syntaxisfouten, in zowel de objectcode als metacode, in een sjabloon worden gevonden tijdens het ontleden.

Enkel ontleden van een sjabloon is nog geen garantie dat gegenereerde code vrij zal zijn van syntaxisfouten. De sjabloonevaluator dient uitgerust te zijn met een mechanisme om dit te garanderen. Ons mechanisme werkt op de volgende wijze. Het ontleden van een sjabloon resulteert in een ontleedboom. Deze boom bevat subbomen van de objectcode en subbomen van de metacode. Tijdens evaluatie worden de subbomen van de metacode vervangen door objectcodesubbomen. Hierbij controleert de sjabloonevaluator of de top van de nieuwe objectcodesubboom hetzelfde type heeft als de top van de metacodesubboom. Als dit het het geval is, mag de objectcodesubboom de metacodesubboom vervangen. In het andere geval wordt een accurate foutmelding gegenereerd. Het resultaat is een ontleedboom van de objecttaal en dus syntactisch correct. Deze eigenschap noemen we *syntaxisveilig*.

Om te valideren dat de gepresenteerde technieken de onderhoudbaarheid verbeteren en syntactische correctheid garanderen, hebben we onze ideeën geïmplementeerd in een syntaxisveilige sjabloonevaluator genaamd *Repleo*. Repleo is gebruikt voor een viertal casussen. De eerste casus belicht de generatie van een gebruikelijk situatie waar een webapplicatie op basis van drie lagen gegenereerd wordt uit een datamodel. Deze casus toont dat meerdere lagen van de applicatie gebaseerd op verschillende programmeertalen kunnen worden gegenereerd vanuit hetzelfde model. De tweede en derde casus zijn gebruikt om aan te tonen dat onze metataal leidt tot verbeter onderhoudbare codegeneratoren. Onze metataal dwingt het af om een codegenerator op te bouwen in twee strikt gescheiden lagen, waar de originele codegeneratoren minder modulair zijn. De laatste casus toont aan dat syntaxisveiligheid resulteert in het vergroten van de veiligheid van dynamisch gegenereerde webpagina's.

Om terug te komen op correctheid van de gegenereerde code. We hebben ook laten zien dat het mogelijk is om de statisch semantische eigenschappen van sjablonen te checken. Statisch semantische checks zijn gedefinieerd voor de metataal, de objecttaal en checks voor situaties waar de objecttaal afhankelijk is van de metataal. We hebben een prototype van een statisch semantische checker voor PicoJava sjablonen geïmplementeerd op basis van attribuut grammatica's. Het gebruik van attribuut grammatica's heeft geleid tot hergebruik van de originele PicoJava checker.

Tot slot, in dit proefschrift hebben we de eisen voor een metataal opgesteld en besproken hoe een syntaxisveilige sjabloonevaluator kan worden geïmplementeerd. Dit resulteert in beter onderhoudbare codegeneratoren en betrouwbaardere gegenereerde code.

Curriculum Vitae

PERSONALIA

Bastiaan Jeroen Arnoldus
Geboorteplaats: Amstelveen, Nederland
Geboortedatum: 9 oktober 1981

OPLEIDINGEN

Software Engineering (doctorandus) (2004-2005)
Universiteit van Amsterdam, Amsterdam
Diploma behaald in 2005

Elektrotechniek (ingenieur) (2000-2004)
Hogeschool van Amsterdam, Amsterdam
Propedeuse behaald in 2001
Diploma behaald in 2004

Gymnasium (1994-2000)
Keizer Karel College, Amstelveen
Diploma behaald in 2000

WERKERVARING

Hogeschool van Amsterdam, Amsterdam (2005-heden)
Promovendus
Domein Media, Informatie en Creatie
Opleidingen Informatica en Technische Informatica

BJA Electronics, Amstelveen (2005-2010)
Eigenaar
Ontwikkeling van maatwerk elektronica en software
voor theater en musicalproducties

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking*

Timed Automata. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02