To Reuse
or
To Be Reused

Techniques for Component Composition and Construction

Merijn de Jonge

To Reuse or To Be Reused

Techniques for Component
Composition and Construction

To Reuse or To Be Reused

Techniques for Component
Composition and Construction

Promotor: prof. dr. P. Klint
Co-promotor: dr. A. van Deursen
Faculteit: Natuurwetenschappen, Wiskunde en Informatica

Preface

This thesis presents results of four years of research that I performed at the CWI and the University of Amsterdam (UvA). I started this research in 1998 at the UvA with a project on pretty-printing (or *happy-printing* as Margreet Hovenkamp prefers to call it). The project was concerned with developing generic pretty-print technology, and with integrating it in the ASF+SDF Meta-Environment. Although I did develop the technology (which is presented in Chapter 4), happy printing is still not a feature that the ASF+SDF Meta-Environment supports. The reason is that reusing the pretty-print technology in different applications (such as the ASF+SDF Meta-Environment) turned out to be rather complicated.

Later, I realized that ever since my Master's, I have been interested in this reuse aspect of computer science. Therefore, it comes as no surprise that software reuse forms the connection between my research projects. In this thesis I present the techniques developed in these projects and I discuss how they help to improve software reuse. Now they only have to be applied, in order to integrate the pretty-print technology in the ASF+SDF Meta-Environment ...

Many people contributed to the development of this thesis. Below I would like to thank them.

First of all, I would like to thank Paul Klint, my promotor, who has offered me a creative and inspiring research environment, both at the UvA and the CWI. We first met when I was considering to move to the UvA, halfway my study. He made this decision a very easy one. The move to Paul's group, awakened my general interest in academic research. In particular, my interest in computer science was born, which inspired me in writing this thesis.

I am also grateful to my co-promotor, Arie van Deursen, who taught me almost anything about writing good research papers. He assisted me during the writing of most of the articles on which this thesis is based. I enjoyed the pleasant discussions we had and appreciate his never-ending enthusiasm for my ideas. We also wrote an article together (see Chapter 7) and I hope to continue our cooperation in the future.

I thank the members of the reading committee: dr.ing. Krzysztof Czarnecki, prof.dr. Jan Bosch, dr. Frank van der Linden, prof.dr. Jan Bergstra, prof.drs. Maarten Boasson, and prof.dr. Peter Sloot, for carefully refereeing this thesis.

Joost Visser was my room mate at the CWI, as well as my travel mate during several conferences, workshops, and summer schools. It was great to discover countries, cities, and hills together. Furthermore, he was my squash mate, at all times that we needed to break out, to discuss work and to make new plans. Of course, at summer time, we preferred to enjoy the sun and played tennis. Last but not least, he was my work mate. He co-authored Chapters 2 and 3, but we also had plenty of other joint projects. I hope we will continue our cooperation in the future.

Leon Moonen was my second room mate. I will never forget our hike in Ireland. How we almost got lost in the rain and the mist, and how we had dinner in the home-restaurant of our hostess. I enjoyed all discussions about work and non-work, and I really regret that, despite of our joint interests, we never wrote an article together. But there is hope: project *M335* is still alive . . .

Eelco Visser has been an inspiring colleague who has supported my ideas with much enthusiasm. He always had a solution at hand for any of my research problems. Together with Joost Visser, we initiated XT (see Chapter 3), perhaps the first bottom-up strategy for promoting the programming language Stratego. I am also proud that I was Eelco Visser's first Master's student.

Ramin Monajemi has been a close friend and colleague since we became partners as student assistants. We visited many of Amsterdam's pubs, where we happily consumed music, beers and life. We both like cycling and I enjoyed our cycling tour from Deventer to Amsterdam. Next trip from Amsterdam to Nijmegen? During his contract at Lucent Technologies, we did a project together, which allowed me to bring academic work (in the form of the XT bundle) into practice. As a result, we wrote Chapter 5 of this thesis together.

I enjoyed working with Tobias Kuipers. We met shortly after my move to the UvA, when he, together with Leon Moonen, was a student assistant. In addition to Paul Klint, they too, rouse my passion for computer science. After my Master's we became colleagues and we did some joint projects and journeys. Our trip, together with Joost Visser, through the middle of Portugal was great fun, and the Englishman shouting: "It's a dead end", is unforgettable. One of our joint projects became a published paper and is included as Chapter 7 in this thesis. It was co-authored with Arie van Deursen.

The rest of the group at the CWI helped to make the stimulating environment that I enjoyed so much. Here I would like to thank them. Mark van den Brand for the discussions about pretty-printing and for his enormous effort in making the ASF+SDF Meta-Environment to what it is today. Jurgen Vinju for his cheerfulness, enthusiasm, and his effort to make all our software *good*. Pieter Olivier and Hayco de Jong for making the ATERMS library, on which most, if not all, of the software developed in our group is based. Jan Heering, for being critical and objective. Ralf Lämmel, for his interest in my work. It is a pity that we never did something together.

My brother Joost de Jonge and my good friend Maarten Noordijk deserve special mention for their help as my 'paranimfen'.

I would like to thank my friends and family, for their support, good times, and confidence. In particular my parents, Els and Frits, for always stimulating me, and for providing a residential hotel whenever I needed one.

Finally, I would like to thank Petra for her love and for being with me. Many thanks for her help during the last phase of this thesis, in keeping me motivated and for giving me inspiration. Now that we finally live together, I am able to thank her every day.

<div style="text-align: right;">

Merijn de Jonge

Amsterdam
January 2003

</div>

Contents

III Epilogue   137

Introduction

1.1 Software reuse

Software reuse is a means to improve the practice of software engineering by using existing software artifacts during the construction of new software systems [92]. Reuse aims at increasing the productivity and quality in large-scale software development [130]. The productivity of software development can be increased because for the development of a new system not all software needs to be developed from scratch but existing artifacts can be used (as-is) [103]. The quality of software can be increased because "proven" technology can be reused [73].

Software reuse is not limited to source code fragments, but may include documentation, specification, design structures and so on [61, 92]. In this thesis we concentrate on reuse of source code fragments and of pre-compiled units such as executable programs and libraries.

The fundamental unit of software reuse is the component [11]. Components can be used in different contexts and compositions to form different software systems, giving rise to component-based software development. For example, in Figure 1.1(a) the architecture of a component-based system called the ASF+SDF Meta-Environment is depicted. This is an environment for language prototyping and for the construction of program transformations [27]. Concepts in this application domain include parsing, pretty-printing, compiling, and debugging.

The clear separation of functionality in the ASF+SDF Meta-Environment ensures that its components can also be used to build additional systems with. Typical applications in this domain require parsing and pretty-printing and can reuse the parse and pretty-print components from the ASF+SDF Meta-Environment. For instance, the program transformation depicted in Figure 1.1(b) first parses its input, then performs the transformation (elimination of goto
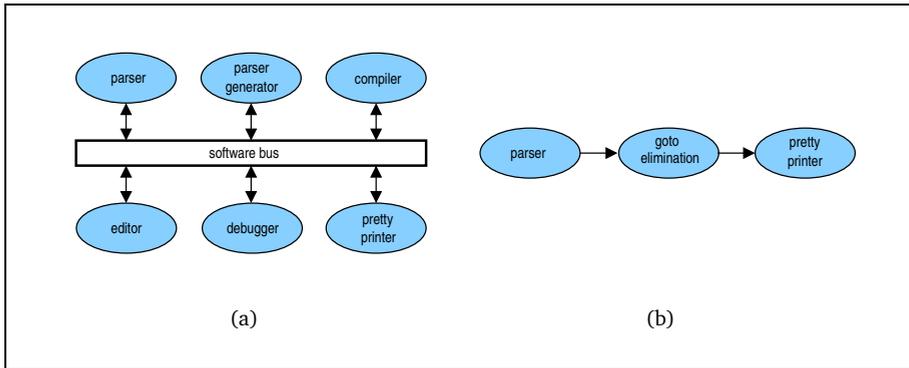
Figure 1.1 Examples of component-based software systems. (a) Architecture of the ASF + SDF Meta-Environment [27]. (b) Architecture of a component-based program transformation.

statements in the example), and finally it transforms the resulting program to plain text using a pretty-printer. This application can be constructed by reusing the parse and pretty-print components from the ASF + SDF Meta-Environment. The goto elimination itself is then the only application-specific component that has to be developed.

An ultimate goal of software reuse is the rise of a large component industry that delivers reusable, high-quality, well-tested components. Software construction then becomes a collaborative development activity because different parts of a system are developed by different people at different institutes.

In 1968 McIlroy was the first to recognize this and to distinguish *manufacturers* which are producers of reusable components and *system builders* that use them [103]. He suggested *mass-produced* software components by a software component sub-industry consisting of *software manufacturers* dedicated primarily to the development of reusable software components. This distinction of manufacturers and system builders yields two complete development cycles: development *for* reuse and development *with* reuse. The first cycle is focused on developing families of systems rather than one-of-a-kind systems, the second development cycle is concerned with building family members [51].

Despite its attractiveness, software reuse is difficult in practice [92, 11, 69]. Software construction with mass-produced software components, for instance in the form of Commercial Off-The-Shelf (COTS) components, as well as collaborative software development are therefore not common practice yet. Software reuse is difficult because it is hard to satisfy simple requirements on software reuse. Krueger distinguishes four such requirements (which he calls *reuse truisms*) [92]:

1. An effective reuse technique must reduce the cognitive distance between an initial concept and its final executable implementation. That is, it must provide proper abstractions for reusable artifacts.

2. It must be easier to reuse an artifact than to develop it from scratch.

3. To select an artifact for reuse, you must know what is does.

4. Finding a reusable artifact must be faster than developing it from scratch.

Truisms 1, 3, and 4 require a proper abstraction mechanism in order to obtain a conceptual understanding of reusable artifacts. Truism 2 is concerned with technical aspects that simplify software construction from individual components.

To make software reuse more successful, techniques are needed that assist manufacturers in building reusable software components, and system builders in finding, selecting, and integrating them in composite software systems. The above reuse truisms can be used to evaluate such techniques in order to judge their effectiveness. Software reuse techniques involve 'abstraction', 'component composition', and 'component granularity'. These are the central themes of this thesis and will be discussed in more detail in the next sections.

1.2 Abstraction

A component is an abstraction consisting of an abstraction specification or *interface* that is externally visible, and an abstraction realization or *implementation* that is hidden [92, 11, 116, 36]. Observe that this is a much broader definition than the one given in [132], where components are defined as *binary* units.

Abstractions are hard to define for generally reusable artifacts because we do not have many universal abstractions available that go beyond the abstraction level of stacks, lists, trees etc. [19, 44, 92]. Consequently, the cognitive distance of such *domain-independent* abstractions is high and the payoff for reusing them is relatively small.

On the other hand, software reuse can be successful in case it is *domain-specific* and the domain provides proper domain concepts for reusable artifacts (typical one-word idioms) [132, 108]. Examples are math libraries for developers familiar with mathematical concepts, and domain-specific application generators. These domain concepts describe artifacts in terms of "what" they do rather then "how" they do it and allow a software developer to reason in terms of these abstractions [92].

The functionality of a software component is usually not fixed. Rather, to improve its usability, it is often adaptable for specific needs. A component interface therefore consists of a variable part and a fixed part. The variable part corresponds to possible variants in the component's implementation and maps to the collection of possible implementations, the fixed part expresses invariant characteristics of the component [92]. Examples of such invariants are the (fixed) parse algorithm used in a parse component (such as LR(1) parsing), or the maximal line length that the parser accepts as input. An example of a

possible variant is the error routine that should be called by the parse component to report syntax errors. Instantiating the variable part of a component corresponds to component configuration.

Combining components to form a software system implies combining their fixed and variable parts. Combining the variable parts may easily lead to a combinatorial explosion of possible configurations. Many of these may not be needed for the composite system, may not be useful, or not be meaningful (i.e., semantically incorrect) [51, 11].

As an example, assume the goto elimination of Figure 1.1(b) is used in a larger transformation framework where it must be combined with additional transformations. The variable parts of the three components of the goto elimination must then be combined with all the variable parts of all other transformations in the framework. Depending on the number of transformations in the framework this leads to complicated configuration.

Clearly, such component compositions also require abstractions. The variable parts of these abstractions are subsets (or sensible combinations) of the individual variable parts at a higher level of abstraction. For instance, the ASF+SDF Meta-Environment is an abstraction for the composition of the six components parser, parser generator, compiler, editor, debugger, and pretty-printer. It will (partially) instantiate the variable parts of these components and it will have a variable part at a higher level of abstraction than these individual components.

Such abstractions are called *layered abstractions* [92] because the abstraction specification of one layer forms the implementation of the next higher layer. A challenge is to make layered abstractions compositional such that new layers can easily be constructed [111]. Although various approaches exist (e.g., GenVoca [11], Koala [112]), there is a need for more general, language-independent solutions. Moreover, configuration validation, for instance by modeling configuration constraints, is needed to automatically detect and prevent invalid component configurations [51, 8, 9].

Abstractions for component compositions can be domain-specific and are either technical or consumer-related. The group of products (or systems) that can be built from technical abstractions forms a *product family* [115] (or system family). The group of products that can be built from consumer-related abstractions forms a *product line*. These consumer-related abstractions have a non-technical nature and correspond to the specific needs of a selected market. Thus, a product line is based on marketing strategy rather than on technical similarities between products [51]. Observe that a product line need not be a product family, although that is how its greatest benefits can be achieved [45, 51].

For example, the components in Figure 1.1 are abstractions in the domain of language processing. The corresponding product family includes the ASF+SDF Meta-Environment and software renovations like goto elimination. A typical product line would be a COBOL transformation factory, supporting the features

goto elimination and copybook expansion. Individual product instances can be configured to feature one or more of these.

The abstractions used in product families constitute the *problem space*. The variability of a product family is called the *configuration space* and defines the possible group of products (i.e., its family members). Specifying individual family members by instantiating the variable part of a product family is performed using terminology (or abstractions) in the problem space. The *solution space* contains the corresponding implementation components of a product family together with their possible configurations [51].

Components implement an *abstract-to-concrete* mapping [11], or, in the terminology of [92], each abstraction specification has an abstraction realization (i.e., implementation). The same holds for layered abstractions. Consequently, the abstraction specification of a product family, constituting the problem space, has a realization in the solution space.

A challenge is to automate this abstract-to-concrete mapping such that an implementation can automatically be derived from a configuration in the problem space. *Generative programming* is a software engineering paradigm that aims at this automated mapping [51].

1.3 Component composition

With software component reuse, software systems become *composite* systems (i.e., collections or compositions of application-specific and reusable components [11]), instead of monolithic systems. The functionality of such systems is spread over the individual components and needs to be integrated to obtain the desired behavior of the composite system.

Components that form a system thus function as building blocks and should be designed for integration. Integration can occur at different moments in time, each requiring a different integration mechanism. Some integration moments that can be distinguished on this integration time line include:

Development-time integration It is concerned with assembling reuse repositories containing all source modules of the components that constitute a composite software system. Source integration is a technique for assembling such reuse repositories and will be discussed in more detail below.

Pre compile-time integration It is concerned with merging reusable functionality in the source code of the system under construction. The resulting source can benefit from the type system of the programming language being used, and from source code level optimizations. Pre compile-time time integration may therefore reduce run-time overhead due to method invocations of small reused functionality. By combining it with layers of abstractions, it can help to reduce the difficulty of scaling reuse libraries in size and feature variants (i.e., the library scaling problem [18]). A promising technique for pre compile-time integration is Aspect Oriented

Programming (AOP), which is a technique to weave functionality (aspects) at explicit positions in source code (join points) [88].[1]

Compile-time integration Compile-time integration is the traditional way of reusing functionality in applications. Reusable code is stored in libraries and linked with application-specific code to the final executable application. The functionality is accessed using function or method invocations. This kind of integration is language-specific and makes integration of components implemented in different languages difficult. Systems implemented in strongly typed languages can benefit from the type system to assure that the functional composition is valid.

Distribution-time integration Component integration at distribution-time is concerned with packaging the components that form an application such that it can be distributed as a unit, and with the installation process of the application. This is also referred to as 'content delivery' [41]. Components can be distributed in either source or binary form. If components are distributed in source form, then distribution-time integration should also address building the composite system. Package managers, such as RPM [6], are often used to build distributions of applications and to install the applications on computer systems.

Run-time integration Components in the form of executable programs or dynamic loadable libraries can be integrated at run-time. A standard example is the Unix programming environment, where little tools, each designed to perform a simple task, can be combined to form advanced programs [87]. Integration in the Unix environment usually takes place in pipelines without type checking. More advanced run-time integration techniques are offered by component architectures such as COM [24, 124], CORBA [109], and EJB [102], or coordination architectures such as the TOOLBUS [16]. Functionality is accessed via message passing and type checking is based on component interface definitions, i.e., signatures that define the services offered by a component. Language-independence is an important benefit of run-time integration, although it is not supported by all run-time integration mechanisms.

As an example, Figure 1.1 shows the composition of reusable components in two different systems. The ASF + SDF Meta-Environment in Figure 1.1(a) is an interactive system that interacts with its user via a graphical user interface. The 'goto elimination' transformation depicted in Figure 1.1(b), on the other hand, is non-interactive. It transforms programs without further user interaction. The components of the ASF + SDF Meta-Environment are therefore integrated via a bus architecture. Communication between components can take place in any

---

[1]Observe that weaving in AOP is not restricted to compile-time, but that it can occur at any time, even at run-time.

order and is accomplished by sending messages over the bus. For the transformation system, a pipeline architecture is used because all communication between components takes place in a fixed left-to-right direction (the output of a component to the left, forms the input of the component to its right).

Despite these conceptually different integration techniques, the figure does not show *how* these components are integrated and *when* they have been integrated. For instance, the pipeline might be implemented at run-time using Unix pipes, or at compile-time using the functional composition:

$$pretty\text{-}print(goto\text{-}elimination(parse(input)))$$

Components are most often designed with a single integration mechanisms in mind. But for the construction of composite systems all integration mechanisms can be combined. To make component-based software development successful, it should not be difficult to construct composite software systems from a wide range of components with different integration techniques. To that end, component interfaces [11, 116, 14] and standardized exchange formats are essential. Component interfaces serve to make software components interchangeable (plug compatible) by hiding their implementations. Standardized exchange formats are inevitable to easily integrate different types of components (such as executable programs or library functions) anywhere on the integration time line and independently of an implementation language.

In addition to the integration techniques discussed thus far, which are concerned with functional integration, *source integration* is another technique that is important for successful software reuse. It is performed at development-time, in advance of all other integration techniques and is concerned with merging all source modules, all build instructions, and all compile-time configuration of the components that constitute a software system.

Source integration is the opposite of decomposing a software system in reusable, independent components. From a software engineering perspective, decomposition complicates the software engineering process, because an application built from individual pieces is organized as a collection of components rather than as a single unit. Consequently, it is hard to develop, maintain, configure, and distribute such systems as a whole. The purpose of source integration is to improve this situation by merging the source modules of reused components, as well as corresponding configuration knowledge and build instructions, to reconstitute a single unit.

Source integration is of particular importance when software reuse extends project or institute boundaries [83]. Typical examples are reuse of commercial off-the-shelf (COTS) source components and open source software reuse [37]. To promote such "third-party" software reuse, source integration techniques including release management [71] and proper abstraction mechanisms in the form of *source code components*, are essential.

A challenge of component composition is to automatically obtain all components that constitute a system, to configure them properly, and to assemble

the software system from them. Knowing how to make these components fit together is another key challenge.

1.4   Component granularity

The granularity of a component is not well defined (e.g., it can be a function, a module, or a complete software system), but it affects two important properties of a component: the payoff or benefit that is gained by reusing the component, and the general usefulness of the component. Development of reusable software components (development for reuse) may therefore serve two different goals:

- Increasing the ratio of reused versus newly developed software (i.e., the *reuse level* [49, 118]) of composite software systems by developing components that provide high payoff.

- Increasing the reuse of individual components by developing components of general usefulness.

These goals can be formulated as: "to reuse or to be reused".

To meet the first goal (increasing the reuse level), large collections of reusable components, providing high payoff to programmers using them, should be available and easily accessible. Payoff, i.e., less lines of code that need to be written, can be increased by using large-scale components [18, 111].

Unfortunately, large-scale components tend to be more specialized for the application domain (i.e., domain-specific). Consequently, the probability of being reused decreases as components increase in size [18, 111, 131]. Another problem is that large-scale components may themselves include more general functionality, which does not come available for reuse outside the component [69].

Thus, to meet the second goal (increasing the reuse of individual components), components should be made more generally applicable by restricting their size and reducing their functionality.

For example, Figure 1.1 shows examples of large-scale reusable components in the domain of language processing (e.g., parsers and compilers). The coarse-grained granularity of these high-level components hides lower level componentising with less domain specificity. Since one might expect that commonalities also exist between the components within each application (for instance, for data exchange and communication in case of the ASF + SDF Meta-Environment), the granularity of software reuse depicted in the figure is not optimal. To achieve fine-grained software reuse, components should be split in smaller reusable units, which have more general purpose applications. As an example, Figure 1.2, shows a more detailed view of the ASF + SDF Meta-Environment with fine-grained software reuse.
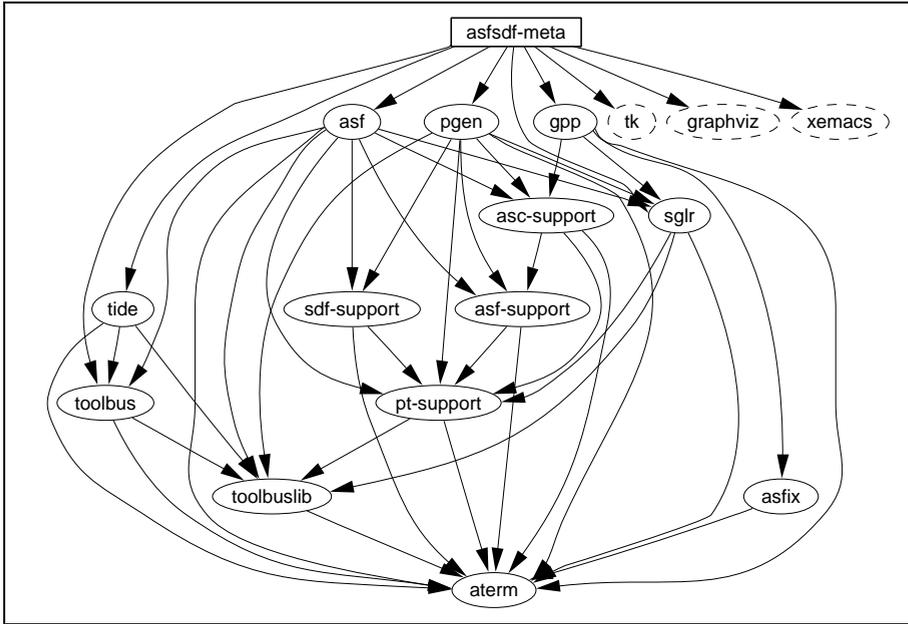
Figure 1.2 Architecture of the ASF + SDF Meta-Environment at the implementation level with extensive, fine-grained software reuse *between* functional components. Nodes correspond to source code components, arrows denote reuse relations, dashed nodes denote third-party components.

It is not difficult to imagine that the development process of an application as composition of many small components is considerably more complicated than an application assembled from a few large-scale, domain-specific components. The reason is that large-scale components provide higher payoff for application builders in terms of lines of code to write [18]. Furthermore, application builders can benefit from large-scale component reuse because domain-specific concepts are easier to understand than low-level, generally applicable components [13, 92]. Finally, building, testing, distributing, and deployment are relatively easy for an application consisting of only a single component but become complex activities when the number of components increases.

Apparently, the reuse processes "development for reuse" and "development with reuse" have conflicting goals [13, 108]. While the first process typically would deliver small, flexible, generally applicable components, the latter process demands large-scale, domain-specific components (see Table 1.1). The trade-off between component size and reuse effort yields interesting software engineering challenges. Existing techniques for development of reusable components, such as layered abstractions [11] and domain-specific library development [13], should be combined with generative techniques for automated component integration at the functional and the source code level.

Quality	Granularity	
	Coarse-grained	Fine-grained
Payoff	+	−
Cognitive distance	+	−
Software construction process	+	−
General usability	−	+
Code duplication across components	−	+

Table 1.1 Component granularity affects reuse benefits due to different qualities of coarse-grained and fine-grained components.

The challenge here is to combine both goals by finding good design principles and by developing proper integration and composition techniques. These would allow large-scale components to be decomposed into small, general components which are more widely applicable. These smaller components can be composed and integrated easily to offer benefits of large-scale components.

1.5 Research questions

The objective of this thesis is to develop an architecture for effective software reuse where components can be developed by different people at different institutes, and be integrated easily in composite software systems. To establish such collaborative software development, we distinguish development *for* reuse and development *with* reuse.

Our research therefore concentrates on reuse techniques for both development cycles that satisfy the reuse requirements (reuse truisms) discussed in Section 1.1. These techniques require answers to the following research questions related to abstraction, composition, and granularity.

1.5.1 Abstraction

Domain abstractions improve the reusability of software components because they can reduce the cognitive distance between the initial concept of a system and its final executable implementation [92, 13, 18]. Figure 1.1 shows some large-scale components in the domain of language processing. This suggests that this domain provides proper abstractions for building a family of language tools from high-level reusable components.

> **Question 1**
>
> How can an effective software reuse practice in the domain of language processing be established?

1.5.2 Composition

A productive component market would deliver a wide range of components, designed for different integration mechanisms, programmed in different programming languages, and located at a diverse number of places. True collaborative software development demands that such diverse components can easily be composed, retrieved, and configured. However, in practice achieving such compositionality turns out to be rather complicated.

> **Question 2a**
>
> How can the compositionality of components be improved and the composition process be automated?

Different people and institutes use varying techniques and infrastructure for software development. Potential reusable software components are therefore often entangled in project or institute-specific configuration management (CM) systems [40, 112], or depend on local software. Since standardization in CM systems is lacking [112, 151] and because build processes are often not portable [7], reuse of these components over project and institute boundaries is difficult [83]. This hampers collaborative software development.

> **Question 2b**
>
> How can project and institute-specific dependencies of software components be removed in order to promote collaborative software development?

1.5.3 Granularity

Fine-grained software reuse of many small components helps to reduce code duplication. However, it complicates system understanding [13] since the cognitive distance is high [92]. Furthermore, managing build, configuration, and distribution processes of many small components is complicated. Large-scale components on the other hand, increase code duplication due to commonalities between components, but they provide high payoff, decrease cognitive distance, and simplify software engineering (see Table 1.1).

> **Question 3**
>
> Can the conflicting goals of many, small components (fine-grained reuse) and large-scale components (high payoff and low cognitive distance) be combined?

1.6   Overview

In this thesis we seek answers to the aforementioned research questions concerning abstraction, composition, and granularity. To that end, we develop techniques to facilitate effective software reuse.

The thesis consists of two parts. In the first part (Chapters 2–4) we address "development *for* reuse", which is concerned with developing reusable components. We develop a comprehensive architecture for component-based software development in the domain of language processing and instantiate it with newly developed and existing domain-specific components. The instantiated architecture forms a product family in the domain of language processing.

The second part (Chapters 5–7) addresses "development *with* reuse". It is concerned with building applications from reusable components. We demonstrate how the instantiated architecture effectively reduces development time of complex language tools. Further, we discuss automated construction of self-contained systems from individual source components. Finally, we discuss techniques for designing, implementing, and initiating product lines, as well as for automated assembly of individual product members from feature selections.

Below is a summary of the subjects that will be presented in the subsequent chapters.

Chapter 2, "Grammars as Contracts"   This chapter presents a framework for software reuse in the domain of language processing. The framework is designed to separate development and use of language components. We also present a corresponding model for language tool development which we called Language-Centered Software Engineering (LCSE).

Chapter 3, "XT: a Bundle of Program Transformation Tools"   This chapter discusses a collection of generative components for LCSE which forms an instantiation of the architecture developed in Chapter 2. We discuss the roles of XT's constituents in the development process of program transformation tools, as well as some experiences with building program transformation systems with XT. Furthermore, we discuss a mechanism for collecting reuse statistics, which we use in this thesis to measure the effectiveness of our reuse techniques.

The components that are bundled with XT originate from several research projects. My contributions to XT include: design of XT's architecture, development of techniques for building and distributing XT (this resulted in the technique "Source Tree Composition", discussed in Chapter 6), development of several general-purpose language tools, design and initiation of the Online Grammar Base, development of several SDF grammars, and the development of generic pretty-print technology (see Chapter 4). Appendix A summarizes the components to which I contributed, Appendix B contains a list of additional, third-party components that are bundled with XT.

Chapter 4, "Pretty-Printing for Software Reengineering" Pretty-printing forms an integral part of LCSE. To promote reuse of pretty-print components, generic (i.e., language-independent) and customizable pretty-print technology are needed. In this chapter we present the Generic Pretty-Printer GPP and discuss the techniques that it uses to fulfill requirements in the context of software reengineering. GPP forms a generally reusable pretty-print component in our language-centered architecture and is part of the XT bundle discussed in Chapter 3.

Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages" This chapter discusses LCSE in practice using the techniques and language tool components presented in Chapters 2–4. We discuss grammar reengineering and the construction of a documentation generator for a proprietary language dialect. We show that with LCSE the development process of languages and tools can be shortened and that a decrease in maintenance costs can be achieved.

Chapter 6, "Source Tree Composition" A typical problem of component-based applications is their complicated construction and distribution. These tasks are complicated because the structuring of a system in components usually remains visible at construction and distribution-time. Consequently, each constituent component has to be separately retrieved, compiled, installed and so on.

This chapter solves this problem by merging the source trees of each component to form a self-contained implementation of the system in which the construction and distribution tasks of individual components are combined. This process is called Source Tree Composition.

Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages" Chapter 6 addresses automated assembly and configuration of software systems from low-level, technical source code components. This chapter discusses software assembly at a higher level of abstraction using software product lines, where software products are constructed from consumer-related feature selections.

The chapter addresses variability management, feature packaging, and a generic approach to make instantiated (customer-specific) variability accessible in applications.

Chapter 8, "Conclusions" This chapter formulates answers to the four research questions and it collects overall metrics for the reuse techniques that will be discussed in this thesis.

1.7   Origins of the chapters

Most of the chapters in this thesis were published before as separate papers. We list their origin.

Chapter 2, "Grammars as Contracts", was co-authored with Joost Visser. It was presented in 2000 at the second international conference on Generative and Component-Based Software Engineering (GCSE) in Erfurt, Germany [85].

Chapter 3, "XT: a Bundle of Program Transformation Tools", was co-authored with Eelco Visser and Joost Visser. It was presented in 2001 at the first workshop on Language Descriptions, Tools and Applications (LDTA) in Genova, Italy [84].

Chapter 4, "Pretty-Printing for Software Reengineering", was presented in 2002 at the International Conference on Software Maintenance (ICSM) in Montréal, Canada [80].

Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages", was co-authored with Ramin Monajemi. It was presented in 2001 at the International Conference on Software Maintenance (ICSM) in Florence, Italy [82].

Chapter 6, "Source Tree Composition", was presented in 2002 at the 7th International Conference on Software Reuse (ICSR) in Austin, Texas [81].

Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages", was co-authored with Arie van Deursen and Tobias Kuipers. It was presented in 2002 at the second Software Product Line Conference (SPLC) in San Diego, California [52].

Development for Reuse

Grammars as Contracts

This chapter presents a framework for software reuse in the domain of language processing. The framework is designed to separate development and use of language components. We also present a corresponding model for component-based language tool development, called Language-Centered Software Engineering (LCSE).

Component-based development of language tools stands in need of meta-tool support. This support can be offered by generation of code – libraries or full-fledged components – from syntax definitions. We develop a comprehensive architecture for such syntax-driven meta-tooling in which grammars serve as contracts between components. This architecture addresses exchange and processing both of full parse trees and of abstract syntax trees, and it caters for the integration of generated parse and pretty-print components with tree processing components.

We discuss an instantiation of the architecture for the syntax definition formalism SDF*, integrating both existing and newly developed meta-tools that support* SDF*. The* ATERM *format is adopted as exchange format. This instantiation gives special attention to adaptability, scalability, reusability, and maintainability issues surrounding language tool development. The work presented in this chapter was published earlier as [85].*

2.1 Introduction

A need exists for meta-tools supporting component-based construction of language tools. Language-oriented software engineering areas such as development of domain-specific languages (DSLs), language engineering, and automatic software renovation pose challenges to tool-developers with respect to adaptability, scalability, and maintainability of the tool development process.
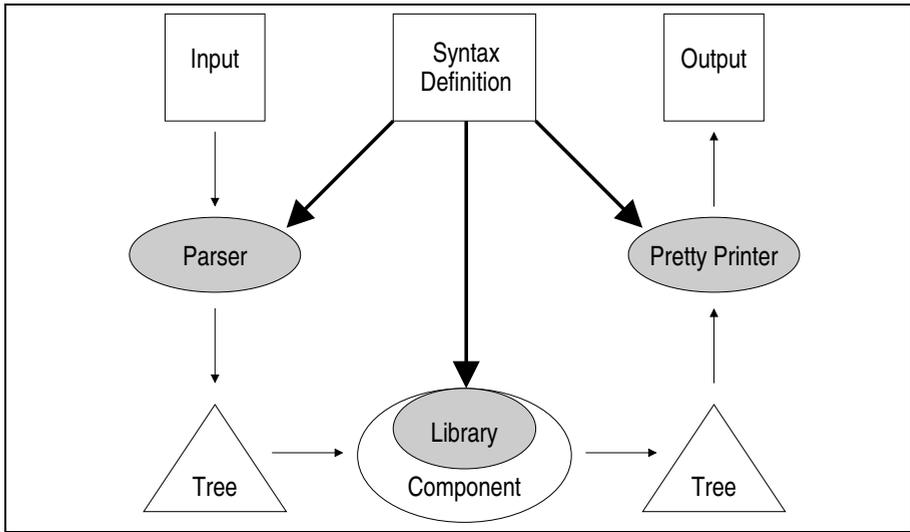
Figure 2.1 Architecture for meta-tool support for component-based language tool development. Bold arrows are meta-tools. Grey ellipses are generated code.

These challenges call for methods and tools that facilitate reuse. One such method is component-based construction of language tools, and this method needs to be supported by appropriate meta-tooling to be viable.

Component-based construction of language tools can be supported by meta-tools that generate code – subroutine libraries or full-fledged components – from syntax definitions. Figure 2.1 shows a global architecture for such meta-tooling. The bold arrows depict meta-tools, and the grey ellipses depict generated code. From a syntax definition, a parse component and a pretty-print component are generated that take plain text into trees and *vice versa*. From the same syntax definition a library is generated for each supported programming language, which is imported by components that operate on these trees. One such component is depicted at the bottom of the picture (more would clutter the picture). Several of these components, possibly developed in different programming languages can interoperate seamlessly, since the imported exchange code is generated from the same syntax definition.

In this chapter we will refine the global architecture of Figure 2.1 into a comprehensive architecture for syntax-driven meta-tooling. This architecture embodies the idea that grammars can serve as contracts governing all exchange of syntax trees between components and that representation and exchange of these trees should be supported by a common exchange format. We call the software engineering process using this architecture *Language Centered Software Engineering* (LCSE). An instantiation of the architecture is available as part of the transformation tools package XT which will be described in the next chapter.

This chapter is structured as follows. In Sections 2.2, 2.3, and 2.4 we will develop several perspectives on the architecture. For each perspective we will make an inventory of meta-languages and meta-tools and formulate requirements on these languages and tools. We will discuss how we instantiated this architecture: by adopting or developing specific languages and tools meeting these requirements. In Section 2.5 we will combine the various perspectives thus developed into a comprehensive architecture. Applications of the presented meta-tooling will be described in Section 2.6. Sections 2.7, and 2.8 contain a discussion of related work and a summary of our contributions.

2.2 Concrete syntax definition and meta-tooling

One aspect of meta-tooling for component-based language tool development concerns the generation of code from *concrete* syntax definitions (i.e., grammars). Figure 2.2 shows the basic architecture of such tooling. Given a concrete syntax definition, parse and pretty-print components are generated by a parser generator and a pretty-printer generator, respectively. Furthermore, library code is generated, which is imported by tool components (Figure 2.2 shows no more than a single component to prevent clutter). These components use the generated library code to represent parse trees (i.e. *concrete* syntax trees), and to read, process, and write them. Thus, the grammar serves as an interface description for these components, since it describes the form of the trees that are exchanged.

A key feature of this approach is that meta-tools such as pretty-printer and parser generators are assumed to operate on the same input grammar. The reason for this is that having multiple grammars for these purposes introduces enormous maintenance costs in application areas with large, rapidly changing grammars. A grammar serving as interface definition enables smooth interoperation between parse, pretty-print, and tree processing components. In fact, we want grammars to serve as contracts governing all exchange of trees between components, and having several contracts specifying the same agreement is a recipe for disagreement.

Note that our architecture deviates from existing meta-tools in the respect that we assume that full parse trees can be produced by parsers and consumed by pretty-printers, not just abstract syntax trees (ASTs). These parse trees contain not only semantically relevant information, as do ASTs, but they additionally contain nodes representing literals, layout, and comments. The reason for allowing such concrete syntax information in trees is that many applications, e.g. software renovation, require preservation of layout and comments during tree transformation.

In order to satisfy our adaptability, scalability and maintainability demands, the concrete syntax definition formalism must satisfy a number of criteria. The syntax definition formalism must have powerful support for modularity and reuse. It must be possible to extend languages without changing the grammar
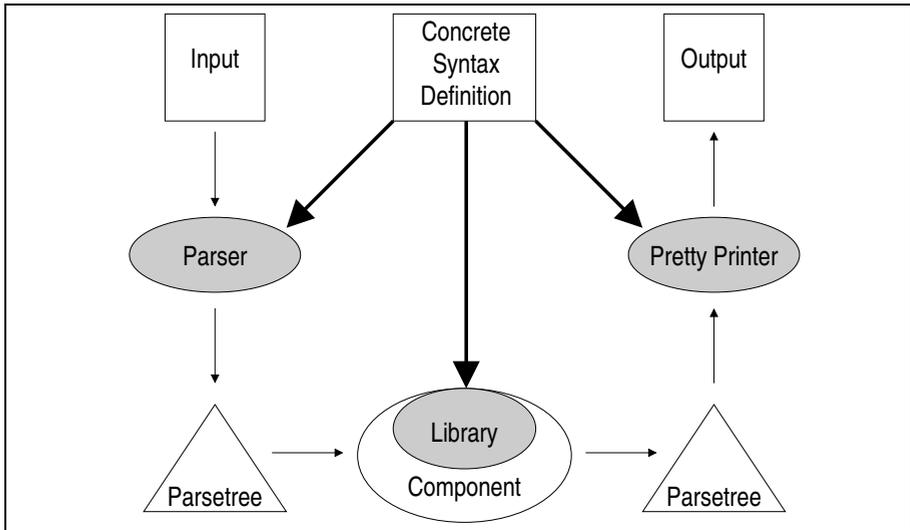
Figure 2.2 Architecture for *concrete syntax* meta-tools. The concrete syntax definition serves as contract between components. Components that import generated library code interoperate with each other and with generated parsers and pretty-printers by exchanging parse trees adhering to the contractual grammar.

for the base language. This is essential, because each change to a grammar on which tooling is based potentially leads to a modification avalanche. Also, the grammar language must be purely declarative. If not, its reusability for different purposes is compromised.

In our instantiation of the meta-tool architecture, the central role of concrete syntax definition language is fulfilled by the Syntax Definition Formalism SDF [68, 137]. Figure 2.3 shows an example of an SDF grammar. This example definition contains lexical and context-free syntax definitions distributed over a number of modules. Note that the orientation of productions is flipped with respect to BNF notation.

2.2.1 Concrete syntax definition

SDF offers powerful modularization features. Notably, it allows modules to be mutually dependent, and it allows alternatives of the same non-terminal to be spread across modules. For instance, the syntax of a kernel language and the syntaxes of its extensions can be defined in separate modules. Also, mutually dependent non-terminals can be defined in separate modules. Renamings and parameterized modules further facilitate syntax reuse.

SDF is a highly expressive syntax definition formalism. Apart from symbol iteration constructors, with or without separators, it provides notation for optional symbols, sequences of symbols, and more. These notations for building

```
definition
module Exp
  exports
    lexical syntax
      [a-z] → Identifier
    context-free syntax
      Identifier                → Exp {cons("var")}
      Identifier "(" {Exp ","}* ")" → Exp {cons("fcall")}
      "(" Exp ")"               → Exp {bracket}

module Let
  exports
    context-free syntax
      let Defs in Exp  → Exp {cons("let")}
      Exp where Defs → Exp {cons("where")}

module Def
  exports
    aliases
      {(Identifier "=" Exp) ","}+ → Defs

module Main
  imports Exp Let Def

  exports
    sorts Exp
    lexical syntax
      [\ \t\n] → LAYOUT
    context-free restrictions
      LAYOUT? -/- [\ \t\n]
```

Figure 2.3 An example SDF grammar.[1]

compound symbols can be arbitrarily nested. SDF is not limited to a subclass of context-free grammars, such as LR or LL grammars. Since the full class of context-free syntaxes, as opposed to any of its proper subclasses, is closed under composition (combining two context-free grammars will always produce a grammar that is context-free as well), this absence of restrictions is essential to obtain true modular syntax definition, and "as-is" syntax reuse.

SDF offers disambiguation constructs, such as associativity annotations and relative production priorities, that are decoupled from constructs for syntax definition itself. As a result, disambiguation and syntax definition are not tangled in grammars. This is beneficial for syntax definition reuse. Also, SDF grammars are purely declarative, ensuring their reusability for other purposes besides parsing (e.g. code generation and pretty-printing).

---

[1]All code examples in this thesis are formatted using the generic pretty-printer GPP, which is described in Chapter 4, "Pretty-Printing for Software Reengineering".

SDF offers the ability to control the shape of parse trees. The alias construct (see module *Def* in Figure 2.3) allows auxiliary names for complex sorts to be introduced without affecting the shape of parse trees or abstract syntax trees. Aliases are resolved by a normalization phase during parser generation, and do not introduce auxiliary nodes.

2.2.2 Concrete meta-tooling

Parsing   SDF is supported by *generalized* LR parser generation [122]. In contrast to plain LR parsing, generalized LR parsing is able to deal with (local) ambiguities and thereby removes any restrictions on the context-free grammars. A detailed argument that explains how the properties of GLR parsing contribute to meeting the scalability and maintainability demands of language-centered application areas can be found in [31]. The meta-tooling used for parsing in our architecture consist of a parse table generator, and a generic parse component, which parses terms using these tables, and generates parse trees [137].

Parse tree representation   In our architecture instantiation, the parse trees produced from generated parsers are represented in the SDF parse tree format, called AsFix [137]. AsFix trees contain all information about the parsed term, including layout and comments (see Figure 2.4). As a consequence, the exact input term can always be reconstructed, and during tree processing layout and comments can be preserved. This is essential in the application area of software renovation.

Full AsFix trees rapidly grow large and become inefficient to represent and exchange. It is therefore of vital importance to have an efficient representation for AsFix trees available. Moreover, component-based software development requires a uniform exchange format to share data (including parse trees) between components. The ATERM format is a term representation suitable as exchange format for which an efficient representation exists. Therefore AsFix trees are encoded as ATERMS to obtain space efficient exchangeable parse trees ([28] reports compression rates of over 90 percent). In Section 2.3.2 we will discuss tree representation using ATERMS in more detail.

Pretty-printing   We use GPP, a generic pretty-printing toolset that will be discussed in Chapter 4, "Pretty-Printing for Software Reengineering". This set of meta-tools provides the generation of customizable pretty-printers for arbitrary languages defined in SDF. The layout of a language is expressed in terms of pretty-print rules which are defined in an ordered sequence of pretty-print tables. The ordering of tables allows customization by overruling existing formatting rules.

GPP contains a formatter which operates on AsFix parse trees and supports comment and layout preservation. An additional formatter which operates on ASTs is also part of GPP.
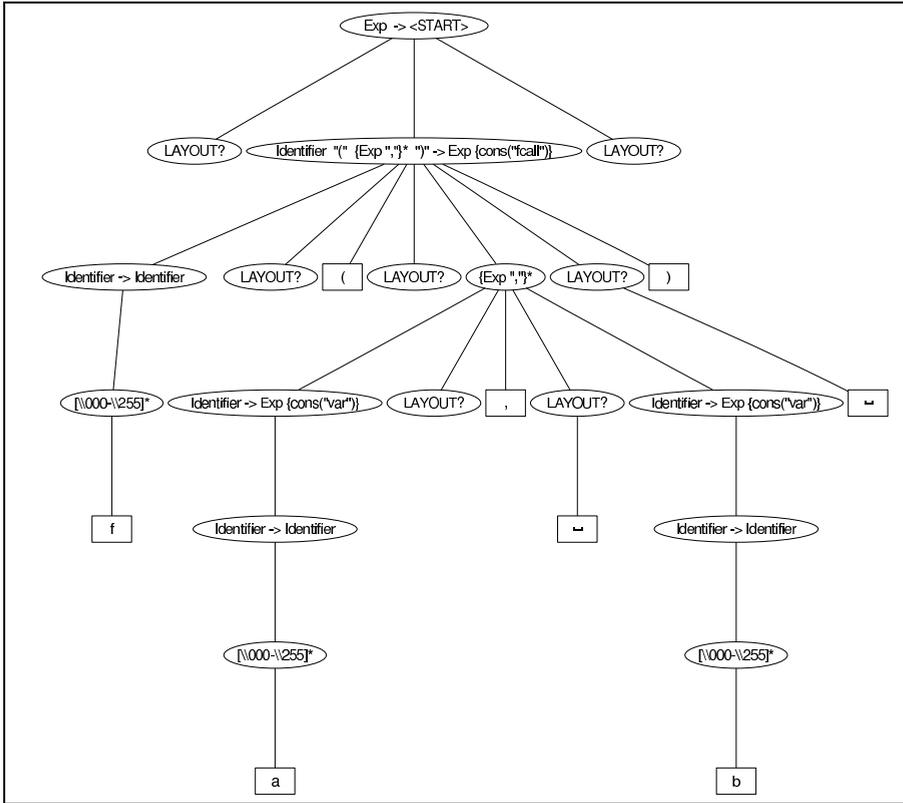
Figure 2.4 Example of a parse tree represented in the SDF parse tree format ASFIX. This figure depicts the parse tree for the expression '$f(a, ␣b␣)$' according to the grammar of Figure 2.3.

GPP is an open system which can be extended and adapted easily. Hence, support for new output formats (in addition to plain text, LATEX, and HTML which are supported by default), as well as language-specific formatters can be incorporated with little effort.

2.3 Abstract syntax definition and meta-tooling

A second aspect of meta-tooling for component-based language tool development concerns the generation of code from *abstract* syntax definitions. Figure 2.5 shows the architecture of such tooling. Given an abstract syntax definition, library code is generated, which is used to represent and manipulate ASTs. The abstract syntax definition language serves as an interface description language for AST components. In other words, abstract syntax definitions serve as tree type definitions (analogous to XML's document type definitions).
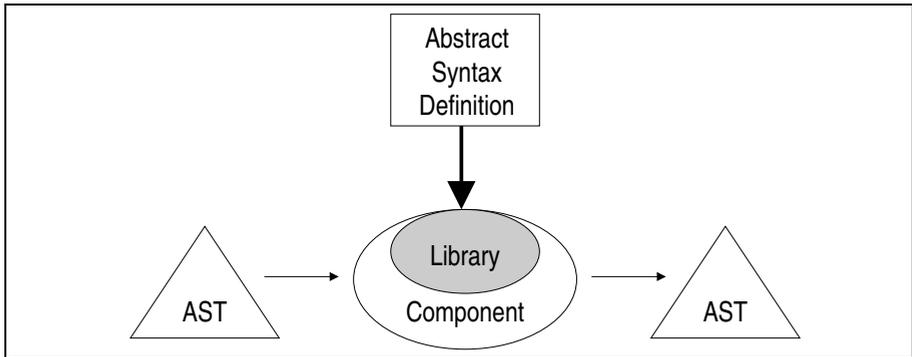
Figure 2.5 Architecture for *abstract syntax* meta-tools. The abstract syntax definition, prescribing tree structure, serves as a contract between tree processing components.

2.3.1 Abstract syntax definition

For the specification of abstract syntax we have defined a subset of SDF, which we call ABSTRACTSDF. ABSTRACTSDF was obtained from SDF simply by omitting all constructs specific to the definition of *concrete* syntax. Thus, AB-STRACTSDF allows only productions specifying prefix syntax, and it contains no disambiguation constructs or constructs for specifying lexical syntax. AB-STRACTSDF inherits the modularity features of SDF, as well as the expressiveness concerning arbitrarily nested compound sorts. Figure 2.6 shows an example of an ABSTRACTSDF definition.

The need to define separate concrete syntax and abstract syntax definitions would cause a maintenance problem. Therefore, the concrete syntax definition can be annotated with abstract syntax directives from which an ABSTRACTSDF definition can be generated (see Section 2.3.3 below). These abstract syntax directives consist of optional constructor annotations for context-free productions (the "cons" attributes in Figure 2.3) which specify the names of the corresponding abstract syntax productions.

2.3.2 Abstract syntax tree representation

In order to meet our scalability demands, we will require a tree representation format that provides the possibility of efficient storage and exchange. However, we do not want a tree format that has an efficient binary instantiation only, since this makes all tooling necessarily dependent on routines for binary encoding. Having a textual instantiation keeps the system open to the accommodation of components for which such routines are not (yet) available. Finally, we want the typing of trees to be *optional*, in order not to preempt integration with typeless, generic components. For instance, a generic tree viewer should be able to read the intermediate trees without explicit knowledge of their types.

```
definition
module Exp
  exports
    syntax
      "var"(Identifier)        → Exp
      "fcall"(Identifier, Exp*) → Exp
module Let
  exports
    syntax
      "let"(Defs, Exp)      → Exp
      "where"(Exp, Defs) → Exp
module Def
  exports
    aliases
      (Identifier Exp)+ → Defs
module Main
  imports Exp Let Def
```

Figure 2.6 Generated ABSTRACTSDF definition.

ASTs are therefore represented in the ATERM format, which is a generic format for representing annotated trees (see Figure 2.7). In [28] a 2-level API is defined for ATERMS. This API hides a space efficient binary representation of ATERMS (BAF) behind interface functions for building, traversing and inspecting ATERMS. The binary representation format is based on maximal subtree sharing. Apart from the binary representation, a plain, human-readable representation is available.

ABSTRACTSDF definitions can be used as type definitions for ATERMS by language tool components. In particular, the ABSTRACTSDF definition of the parse tree formalism ASFIX serves as a type definition for parse trees (see Section 2.2). The ABSTRACTSDF definition of Figure 2.6 defines the type of ASTs representing expressions. Thus, the ATERM format provides a generic (typeless) tree format, on which ABSTRACTSDF provides a typed view.

2.3.3 Abstract from concrete syntax

The connection between the abstract syntax meta-tooling and the concrete syntax meta-tooling can be provided by three meta-tools, which are depicted in Figure 2.8. Central in this picture is a meta-tool that derives an abstract syntax definition from a concrete syntax definition. The two accompanying meta-tools generate tools for converting full parse trees into ASTs and *vice versa*. Evidently, these ASTs should correspond to the abstract syntax definition, generated from the concrete syntax definition to which the parse trees correspond.
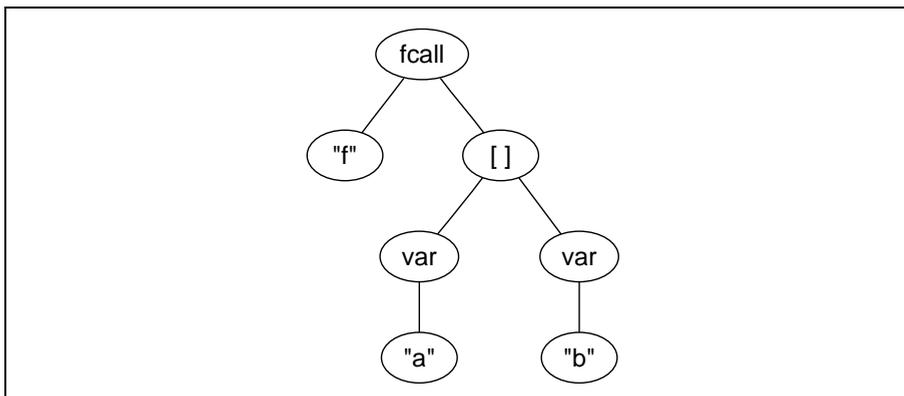
Figure 2.7 Example of an AST. The figure depicts the AST that is derived from the parse tree of Figure 2.4.

An abstract syntax definition is obtained from a grammar in two steps. Firstly, concrete syntax productions are optionally annotated with prefix constructor names. To derive these constructor names automatically, the meta-tool `sdfcons` has been implemented. This tool basically collects keywords and non-terminal names from productions and applies some heuristics to synthesize nice names from these. Non-unique constructors are made unique by adding primes or qualifying with non-terminal names. By manually supplying some seed constructor names, users can steer the operation of `sdfcons`, which is useful for languages which sparsely contain keywords.

Secondly, the annotated grammar is fed into the meta-tool `sdf2asdf`, yielding an ABSTRACTSDF definition. For instance, the ABSTRACTSDF definition in Figure 2.6 was obtained from the SDF definition in Figure 2.3. This transformation basically throws out literals, and replaces mixfix productions by prefix productions, using the associated constructor name.

Together with the abstract syntax definition, the converters `parsetree2-ast` and `ast2parsetree` which translate between parse trees and ASTs are generated. Note that the first converter removes layout and comment information, while the second inserts *empty* layout and comments (see Chapter 4, "Pretty-Printing for Software Reengineering").

Note that the high expressiveness of SDF and ABSTRACTSDF, and their close correspondence are key factors for the feasibility of generating abstract from concrete syntax. In fact, SDF was originally designed with such generation in mind [68]. Standard, YACC-like concrete syntax definition languages are not satisfactory in this respect. Since their expressiveness is low, and LR restrictions require non-natural language descriptions, generating abstract syntax from these languages would result in awkwardly structured ASTs, which burden the component programmers (see Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages").
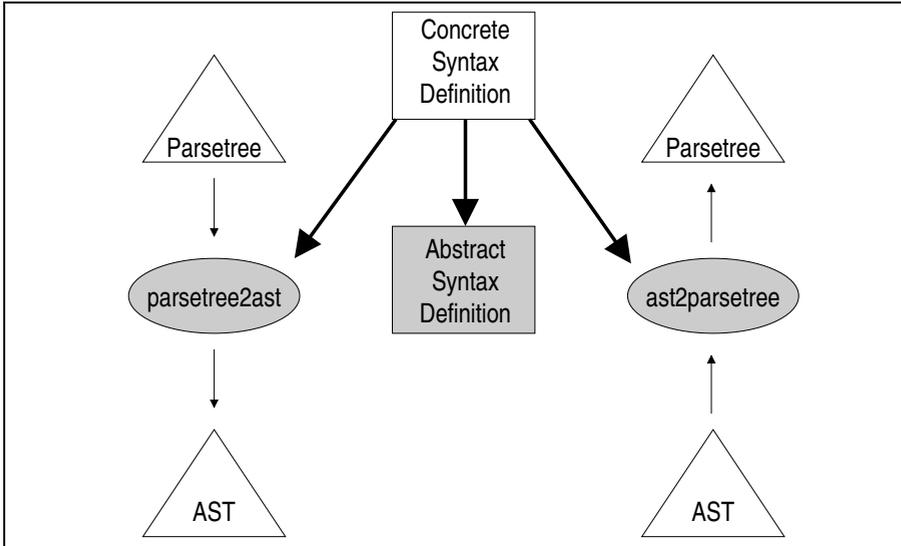
Figure 2.8 Architecture for meta-tools linking abstract to concrete syntax. The abstract syntax definition is now generated from the concrete syntax definition.

2.4 Generating library code

In this section we will discuss the generation of library code (see Figures 2.2 and 2.5). Our architecture for LCSE contains code generators for several languages and consequently allows components to be developed in different languages. Since ATERMS are used as uniform exchange format, components implemented in different programming languages can be connected to each other.

2.4.1 Targeting C

For the programming language C an efficient ATERM implementation exists as a separate library [28]. This implementation consists of an API which hides the efficient binary representation of ATERMS based on maximal sharing and provides functions to access, manipulate, traverse, and exchange ATERMS.

The availability of the ATERM library allows generic language components to be implemented in C which can perform low-level operations on arbitrary parse trees as well as on abstract syntax trees.

A more high-level access to parse trees is provided by the code generator asdf2c which, when passed an abstract syntax definition, produces a library of match and build functions. These functions allow easy manipulation of parse trees without having to know the exact structure of parse trees. These high-level functions are type-preserving with respect to the ABSTRACTSDF definition.

More recently, the generator ApiGen has been developed [77]. It produces

efficient C library code from grammar definitions using a similar approach as asdf2c. It has been used successfully in the development of the ASF+SDF Meta-Environment [27] to eliminate approximately 47% of the amount of handwritten code.

2.4.2 Targeting Java

Also for the JAVA programming language an implementation of the ATERM API exists which allows JAVA programs to operate on parse trees and abstract syntax trees. There is also a code generator for JAVA available which provides high level access and traversals of trees similar to the other supported programming languages. This generator is called JJForester [93] and represents syntax trees as object trees. Tree traversals are supported by generated libraries of refinable visitors. Additionally, JJTraveler [53], provides a JAVA library of generic visitor combinators.

2.4.3 Targeting Stratego

Our initial interest was to apply our meta-tooling to program transformation problems, such as automatic software renovation. For this reason we selected the transformational programming language Stratego [138] as the first target of code generation. Stratego offers powerful tree traversal primitives, as well as advanced features such as separation of pattern-matching and scope, which allows pattern-matching at arbitrary tree depths. Furthermore, Stratego has built-in support for reading and writing ATERMS. Stratego also offers a notion of pseudo-constructors, called *overlays*, that can be used to operate on full parse trees using a simple AST interface.

Two meta-tools support the generation of Stratego libraries from syntax descriptions. Libraries for AST processing are generated from ABSTRACTSDF definitions by asdf2stratego. Libraries for combined parse tree and AST processing are generated from SDF grammars by sdf2stratego. The latter library subsumes the former.[2]

The Stratego code generation allows programming on parse trees as if they were ASTs. Underneath such AST-style manipulations, parse trees are processed in which hidden layout and literal information is preserved during transformation. This style of programming can be mixed freely with programming directly on parse trees. Since Stratego has native ATERM support, there is no need for generating library code for reading and writing trees.

2.4.4 Targeting Haskell

Support for targeting HASKELL is provided by Tabaluga and Strafunski which are discussed in [98, 97]. Code generated in this case is of various kinds.

---

[2]Code generation for Stratego has further been elaborated and applied in [148].

Firstly, the meta-tool `sdf2haskell` generates datatypes to represent parse trees and ASTs. These datatypes are quite similar to the signatures generated for Stratego. Secondly, the DrIFT-Strafunski code generator can be used to generate exchange and traversal code from these datatypes. The generated exchange code allows reading ATERM representations into the generated Haskell datatypes and writing them to ATERMS. The generated traversal code allows composition of analyses and traversals from either updatable fold combinators or functional strategy combinators. We developed the *Haskell ATerm Library* to support input and output of ATERMS from HASKELL types.

Note that not only general purpose programming languages of various paradigms can be fitted into our architecture, but also more specialized, possibly very high-level languages. An attribute grammar system, for instance, would be a convenient tool to program certain tree transformation components.

2.5 A comprehensive architecture

Combining the partial architectures of the foregoing subsections leads to the complete architecture in Figure 2.9. This figure can be viewed as a refinement of our first general architecture in Figure 2.1, which does not differentiate between concrete and abstract syntax, or between parse trees and ASTs.

The refined picture shows that all generated code (libraries and components), and the abstract syntax definition stem from the same source: the grammar. Thus, this grammar serves as the single contract that governs the structure of all trees that are exchanged. In other words, all component interfaces are defined in a single location: the grammar. (When several languages are involved, there are of course equally many grammars.) This single contract approach eliminates many maintenance headaches during component-based development. Of course, careful grammar version management is needed when maintenance due to language changes is not carried out for all components at once.

2.5.1 Grammar version management

Any change to a grammar, no matter how small, potentially breaks all tools that depend on it. Thus, sharing grammars between tools or between tool components, which is a crucial feature of our architecture, is potentially at odds with grammar *change*. To pacify grammar change and grammar sharing, grammar management is needed.

To facilitate grammar version management, we established a *Grammar Base* (see Figure 2.10), in which grammars are stored.[3] Furthermore, we subjected the stored grammars to simple schemes of grammar version numbers and grammar maturity levels.

---

[3]See Appendix A for information about the availability of the Grammar Base.
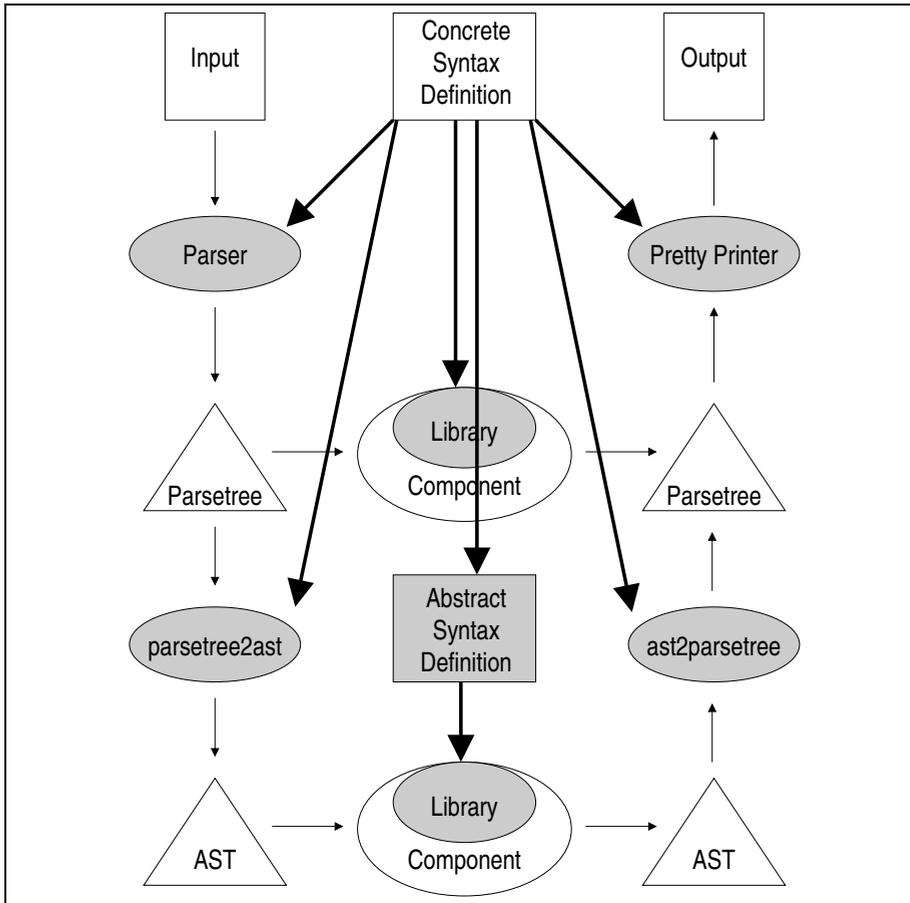
Figure 2.9 Complete meta-tooling architecture. The grammar serves as the contract governing all tree exchange.

To allow tool builders to unequivocally identify the grammars they are building their tool on, each grammar in the Grammar Base is given a name and a version number. To give tool builders an indication of the maturity of the grammars they are using to build their tools upon, all grammars in the Grammar Base are labeled with a maturity level. We distinguish the following levels:

volatile The grammar is still under development.
stable The grammar will only be changed for minor bug fixing.
immutable The grammar will never change.

Normally, a grammar will begin its life cycle at maturity level *volatile*. To build extensive tooling on such a grammar is unwise, because grammar changes are

to be expected that will break this tooling. Once confidence in the correctness of the grammar has grown, usually through a combination of testing, benchmarking, and code inspection, it becomes eligible for maturity level *stable*. At this point, only very local changes are still allowed on the grammar, usually to fix minor bugs. Tool-builders can safely rely on stable grammars without risking that their tools will break due to grammar changes. Only a few grammars will make it to level *immutable*. This happens for instance when a grammar is published, and thus becomes a fixed point of reference. If the need for changes arises in grammars that are stable or immutable, a *new* grammar (possibly the same grammar with a new version number) will be initiated instead of changing the grammar itself.

2.5.2 Connecting components

The connectivity to different programming languages allows components to be developed in the programming language of choice. The use of ATERMS for the representation of data allows easy and efficient exchange of data between different components and it enables the composition of new and existing components to form advanced language tools.

Exchange between components and the composition of components is supported in several ways. First, components can be combined using standard scripting techniques and data can be exchanged by means of files. Secondly, the uniform data representation allows for a sequential composition of components in which Unix pipes are used to exchange data from one component to another. Additionally, the TOOLBUS [16] coordination architecture can be used to connect components and define the communication between them. This architecture resembles a hardware communication bus to which individual components can be connected. Communication between components only takes place over the bus and is formalized in terms of process algebra [4]. Likewise, component architectures such as CORBA [109] can be used.

2.6 Applications

We have used the meta-tooling presented in the previous sections in several projects. We will present a selection of our experiences.

To start with, the meta-tooling has been applied for its own development, and for the development of additional meta-tools that it is bundled with in the Transformation Tools package XT (see Chapter 3). These bootstrap-flavored applications include the generation of an abstract syntax definition for the parse tree format ASFIX from the grammar of SDF. From this abstract syntax definition, a modular Stratego library for transforming ASFIX trees was generated and used for the implementation of some ASFIX normalization components. Also, the tools sdf2stratego, sdfcons, asdf2stratego, sdf2asdf, and

File Edit View Go Communicator   Help

Find more about the Grammar base at the Grammar base home page
http://www.program-transformation.org/gb

Grammars available in version 1.0 of the Grammar Base

(generated on Tue 18 Jun 2002 04:10:32 PM CEST)

Grammar	Version	Maturity	Description
abnf	1	Volatile	ⓘ Augmented BNF for Syntax Specifications
asf	0	Immutable	ⓘ Skeleton grammar of the Algebraic Specification Formalism
asf	1	Volatile	ⓘ Skeleton grammar of the Algebraic Specification Formalism
asfix	2.2	Volatile	ⓘ Parse tree format for SDF
asfix2me	0	Immutable	ⓘ The parse tree format which is used in the ASF+SDF Meta-Environment. This format has been used in completa-0.6 to 0.7.1, and asfsdf-meta0.8 to 0.8.1.
asfix2me	1	Volatile	ⓘ The parse tree format which is currently used in the ASF+SDF Meta-Environment.
aterms	1	Stable	ⓘ Intermediate data representation
aterms	2	Volatile	ⓘ Intermediate data representation
att-sdl	0	Volatile	ⓘ ATs T___
autobundle	0	Volatile	ⓘ G___
bibtex	0	Volatile	ⓘ bi___
box	1	Volatile	ⓘ La___
c	0	Volatile	ⓘ im___
cobol	0	Volatile	ⓘ p___
cobol	1	Volatile	ⓘ p___
cobol-ibm	0	Immutable	ⓘ C___
condperform	0	Volatile	ⓘ C___
condperform	1	Volatile	ⓘ C___
dot	0	Volatile	ⓘ T___
elan			
fdl			

Netscape:

File Edit View Go Communicator   Help

module CICS
imports PROGRAM

exports
  sorts Cics-command Cics-opt Cics-kw Cics-arg

lexical syntax
  [A-Z]+ -> Cics-kw

context-free syntax
  Stat* "EXEC" "CICS" Cics-command Cics-opt* "."   -> Sentence
  "EXEC" "CICS" Cics-command Cics-opt* "END-EXEC" -> Stat
  Cics-kw   -> Cics-opt
  Cics-kw "(" Cics-arg ")"   -> Cics-opt
  A-exp   -> Cics-arg
  Str   -> Cics-arg
  "ADDRESS" "OF" A-exp   -> Cics-arg
  "LENGTH" "OF" A-exp   -> Cics-arg
  "ABEND"   -> Cics-command
  "ADDRESS"   -> Cics-command
  "ASKTIME"   -> Cics-command
  "ASSIGN"   -> Cics-command
  "DELETE"   -> Cics-command
  "DELETEQ" "TS"   -> Cics-command

File Edit View Go Communicator   Help

Generated by toMist (bdejong@cwi.nl)

Language cobol.1

Import Graph

Modules

A-EXP
CICS
COMMENT
COMP
COPY
DATA-BASIS
DATA-DIV
DATA-NAMES
DD-ITEM
DEC

http://www.cwi.nl/~mdejonge/grammar-base/cobol.1/ENV-BASIS.html

Figure 2.10   Screendump of the Online Grammar Base, a collection of browsable, open source SDF grammars. See Appendix A for information about the availability of the Online Grammar Base.

many more meta-tools were implemented by parsing, AST processing in one or more components, and pretty-printing.

Apart from SDF and ABSTRACTSDF, the domain-specific languages BOX (for generic formatting), AUTOBUNDLE (for source tree composition), and BENCH (for generating benchmark reports), have also been implemented with syntax-driven meta-tooling.

The pretty-printer GPP, presented in Chapter 4, "Pretty-Printing for Software Reengineering", has been developed as a collection of reusable components following the LCSE model. GPP reuses the grammars of SDF and BOX to define a grammar for pretty-print tables. Furthermore, components operating on BOX, programmed in different programming languages are combined.

We used LCSE in an industrial project to demonstrate its effectiveness for industrial language tool development. This project, which we discuss in Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages", involved the development of a documentation generator for a proprietary language dialect.

The tool `autobundle` for automated source tree composition has been implemented as a collection of syntax-driven components following LCSE. It is accompanied with additional syntax-driven tooling for web-site generation. These tools are discussed in Chapter 6, "Source Tree Composition".

The generated transformation frameworks for Haskell are being applied to software renovation problems. In [91], a COBOL renovation application is reported. It involves parsing according to a COBOL grammar, applying a number of function transformers to solve a data expansion problem, and unparsing the transformed parse trees. The functional transformers have been constructed by refining a transformation framework generated from the COBOL grammar.

The Stratego meta-tools have been elaborated and applied in the CobolX project [148]. Transformations implemented in this project include data field expansion, and goto-elimination with preservation of layout and comments.

2.7   Related work

Syntax-driven meta-tools for language tool development are ubiquitous, but rarely do they address a combination of features such as those addressed in this chapter. We will briefly discuss a selection of approaches some of which attain a degree of integration of various features.

- Parser generators such as YACC [76] and JavaCC are meta-tools that generate parsers from syntax definitions. Compared with SDF with its supporting tools pgen and sglr, they offer poor support for *modular* syntax definition, their input languages are not sufficiently declarative to be reusable for the generation of other components than parsers, and they do not generally target more than a single programming language.

- The language SYN [23] combines notations for specifying parsers, pretty-printers and abstract syntax in a single language. However, the underlying parser generator is limited to LALR(1), in order to have both parse trees and ASTs, users need to construct two grammars, and code the mapping between trees by hand. Moreover, the expressiveness of the language is much smaller than the expressiveness of SDF, and the language is not modular. Consequently, SYN and its underlying system can not meet our adaptability, scalability and maintainability requirements.

- Wile [150] describes derivation of abstract syntax from concrete syntax. Like us he uses a syntax description formalism more expressive than YACC's BNF notation in order to avoid warped ASTs. Additionally, he provides a procedure for transforming a YACC-style grammar into a more "tasteful" grammar. His BNF extension allows annotations that steer the mapping with the same effect as SDF's aliases. He does not discuss automatic name synthesis.

- AsdlGen [147] provides the most comprehensive approach we are aware of to syntax-driven support of component-based language tools. It generates library code for various programming languages from abstract syntax definitions. It offers ASDL as abstract syntax definition formalism, and *pickles* as space-efficient exchange format. It offers no support for dealing with concrete syntax and full parse trees.

  The choice of target languages, including C and JAVA, has presumably motivated some restrictions on the expressiveness of ASDL. ASDL lacks certain modularity features, compared to ABSTRACTSDF: no mutually dependent modules are supported, and all alternatives for a non-terminal must be grouped together. Furthermore, ASDL is much less expressive. It does not allow nesting of complex symbols, it has a very limited range of symbol constructors, and it does not provide module renamings or parameterized modules.

  Unlike ATERMS, the exchange format that comes with ASDL is always typed, thus obstructing integration with generic components. In fact, the compression scheme of ASDL relies on the typedness of the trees. The rate of compression is significantly smaller than for ATERMS [28]. Furthermore, pickles have a binary form only.

- The DTD notation of XML [34] is an alternative formalism in which abstract syntax can be defined. Tools such as HaXML [146] generate code from DTDs. HaXML offers support both for type-based and for generic transformations on XML documents, using Haskell as programming language. Other languages are not targeted. Concrete syntax support is not integrated.

  XML is originally intended as mark-up language, not to represent abstract syntax. As a result, the language contains a number of inappropriate con-

structs, and some awkward irregularities from an abstract syntax point of view. XML also has some desirable features, currently not offered by AB-STRACTSDF, such as annotations, and inclusion of DTDs (abstract syntax definitions) in documents (abstract terms).

- Many elements of our instantiation of the architecture for LCSE were originally developed as part of the ASF + SDF Meta-Environment [15, 68, 54, 27]. This is an integrated language development environment which offers SDF as syntax definition formalism and the term rewriting language ASF as programming language. Programming takes place directly on concrete syntax, thus hiding parse and abstract syntax trees from the programmers view. Programming, debugging, parsing, rewriting, and pretty-printing functionality are all offered via a single interactive user interface. Meta-tooling has been developed to generate ASF-modules for term traversal from SDF definitions [30, 29].

  The ASF + SDF Meta-Environment offers a single programming language (ASF), programming on abstract syntax is not supported. Support for component-based development is (currently) limited to gluing compiled ASF programs that read and write flat terms.

  To provide support for component-based tool development, we adopted the SDF, ASFIX, and ATERM formats from the ASF + SDF Meta-Environment, as well as the parse table generator for SDF, the parser sglr, and the ATERM library. To these we have added the meta-tooling required to complete the instantiation of the architecture of Figure 2.9. In future, some of these meta-tools might be integrated into the ASF + SDF Meta-Environment.

2.8 Contributions

We have presented a comprehensive architecture for Language-Centered Software Engineering (LCSE). This architecture consists of syntax-driven meta-tooling supporting component-based language tool development. The architecture embodies the vision that grammars can serve as contracts between components under the condition that the syntax definition formalism is sufficiently expressive and declarative, and the meta-tools supporting this formalism are sufficiently powerful. We have presented our instantiation of such an architecture based on the syntax formalism SDF. SDF and the tools supporting it have agreeable properties with respect to modularity, expressiveness, and efficiency, which allow them to meet scalability and maintainability demands of application areas such as software renovation and domain-specific language implementation. We have shown how abstract syntax definitions can be obtained from grammars. We discussed the meta-tooling which generates library code for a variety of programming languages from concrete and abstract syntax def-

initions. Components that are constructed with these libraries can interoperate by exchanging ATERMS that represent trees.

XT: a Bundle of Program Transformation Tools

This chapter discusses XT*, a bundle of program transformation tools. It is a collection of generative components for Language-Centered Software Engineering (LCSE) and forms an instantiation of the architecture developed in the previous chapter.*

The purpose of the XT *bundle is to bundle existing and newly created software components into an open framework for easy development of component-based program transformations. We discuss the roles of* XT*'s constituents in the development process of program transformation tools, as well as some experiences with building program transformation systems with* XT*. Furthermore, we discuss how to measure software reuse in applications built with* XT *components. The work presented in this chapter was published earlier as [84].*

3.1   Introduction

Program transformation encompasses a variety of different, but related, language processing scenarios, such as optimization, compilation, normalization, and renovation. Across these scenarios, many common, or similar subtasks can be distinguished, which opens possibilities for software reuse. To support and demonstrate such reuse across program transformation project boundaries, we have developed XT. XT is a bundle of existing and newly developed libraries and tools useful in the context of program transformation for Language-Centered Software Engineering (LCSE). It bundles its constituents into an open framework for component-based transformation tool development, which is flexible

and extendible. XT is distributed as open source under the GNU General Public License [60].

In this chapter we will provide an overview of XT and an indication of what is possible with it. Section 3.2 fixes terminology and discusses common program transformation *scenarios*. Section 3.3 outlines the program transformation development process that we want to support. Section 3.4 discusses the actual content of the XT bundle, and explains how its constituents can be used to support program transformation development tasks. Section 3.5 summarizes our experiences with XT so far, Section 3.6 discusses measurement of reuse levels, and Section 3.7 wraps up with concluding remarks.

3.2   Program transformation scenarios

Program transformation is the act of changing one program into another.[1] The term *program transformation* is also used for a program, or any other description of an algorithm, that implements program transformation. The language in which the program being transformed and the resulting program are written are called the *source* language and *target* language respectively. Below we will distinguish scenarios where the source language and target language are different (*translations*) from scenarios where they are the same (*rephrasings*).

Program transformation is used in many areas of software engineering, including compiler construction, software visualization, documentation generation, and automatic software renovation. At the basis of all these different applications lie the main program transformation *scenarios* of translation and rephrasing. These main scenarios can be refined into a number of typical *subscenarios*.

Translation   In a *translation* scenario a program is transformed from a source language into a program in a *different* target language. Examples of translation scenarios are synthesis, migration, compilation, and analysis. In program *synthesis* an implementation is derived from a high-level specification such that the implementation satisfies the specification. A prime example of program synthesis is parser generation. In *migration* a program is transformed to another language. For example, transforming a Fortran77 program to an equivalent Fortran90 program. *Compilation* is a form of synthesis in which a program in a high-level language is transformed to a program in a lower-level language. In program *analysis* a program is reduced to some property, or value (i.e., translated to some aspect language). Type-checking is an example of program analysis.

Rephrasing   In a *rephrasing* scenario a program is transformed into a different program in the *same* language, i.e., source and target language are the same.

---

[1]See the Program Transformation Wiki at `http://www.program-transformation.org`.

Examples of rephrasing scenarios are normalization, renovation, refactoring, and optimization. In a *normalization* a program is reduced to a program in a sub-language. In *renovation* a program is changed in order to add new functionality, or to improve some aspect of the program [43]. For example, repairing a Y2K bug. A *refactoring* is a transformation that improves the design of a program while preserving its functionality. An *optimization* is transformation that improves the run-time and/or space performance of the program.

Most program transformations are (intended to be) semantics preserving, although weaker notions of semantics preservation may be appropriate for some scenarios. Renovation, for instance, typically changes semantics to improve behavior of programs.

The list of sub-scenarios is not complete, and in practice many program transformations are a combination of sub-scenarios. For example, a single compiler may perform code optimization after transforming its input to a target language. In fact, XT supports *component-based* development of program transformations, where each component might follow a different transformation scenario.

3.3 Transformation development

The development process of program transformation tools generally consists of the following steps:

1. Obtain (syntax) definitions of the languages involved in the transformation. This may involve grammar engineering (i.e., (re)construction of grammars, transformation of grammars, or assessment of existing grammars; see Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages").

2. Set-up a transformation framework. This may involve reusing generic transformation libraries or generating language-specific transformation libraries, generating parsers, and generating and refining pretty-printers.

3. Design a transformation pipeline. Generally, this pipeline consists of parsers and pretty-printers as front and back-ends, and contains a variety of rephrasing and translation components. The interfaces between the components of the pipeline need to be established in this phase.

4. Implement the components of the pipeline. This involves choosing implementation languages, designing algorithms, and coding.

5. Glue the components to create a complete transformation. For this purpose, common scripting techniques can be used, or more advanced interoperation and communication techniques.

6. Test the individual transformation components and the complete program transformation as a whole.

Of course, iteration over (some of) these steps is often necessary. To aid the developer in constructing program transformation systems, tool support is needed for each of these steps.

3.4 The XT bundle

XT bundles tooling for the construction of program transformation systems. Its purpose is to provide a development environment for LCSE with minimal installation effort, to verify that all components work together, and to provide extensive documentation and instructions about how to use this tooling together. The following tool packages are bundled by XT:

- ATERMS [28] — This is a generic format for representing annotated trees and is used within XT as common tree exchange format to connect individual components to form transformation systems. There are three representations for ATERMS: i) a human-readable, textual representation; ii) a textual representation with subtree sharing; iii) a space efficient binary representation based on maximal subtree sharing. Furthermore, a library of functions for building, traversing, and inspecting ATERMS is available.

- SDF [68, 137] — All grammars bundled with XT are defined in the modular syntax definition formalism SDF. Parsing of arbitrary context-free languages defined in SDF is supported by the parse table generator pgen in combination with the generic parser sglr. The parser generator produces parse tables that are interpreted by sglr using the Scannerless Generalized-LR parsing algorithm.

- GPP— Pretty-printing is supported by the generic pretty-print toolset GPP (see Chapter 4, "Pretty-Printing for Software Reengineering"). It offers language-independent pretty-print facilities based on customizable pretty-print rules to specify the formatting of text. By default, GPP supports plain text, HTML, and LaTeX. The system can be extended easily to support more output formats.

- Grammar Base — The SDF Grammar Base contains a collection of syntax definitions for a growing number of languages, including COBOL, C, XML, SDL, YACC, and JAVA (see Chapter 2, "Grammars as Contracts"). The purpose of the Grammar Base is to offer a reference for language definitions and to provide a collection of open source grammars that can be downloaded for free and are ready for use.

- Grammar Tools — We developed a collection of tools for grammar analysis, grammar (re)construction, and tree manipulation. For example, yacc2sdf (see Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages") *translates* YACC grammars into SDF, and sdfcons (see Chapter 2, "Grammars as Contracts") is a *rephrasing* transformation that adds synthesized constructor names to SDF grammars.

- Stratego [142] — This is a programming language for term rewriting with strategies. It has been used as transformation language for the implementation of many components of XT. An extensive library that comes with the language supports term traversal in many flavors and offers generic language processing algorithms [139].

Program transformation systems can be constructed by connecting components from the different tool packages of XT together. This composition of components (for instance in scripts or pipelines) is simple because all components can be connected to each other via the common ATERMS exchange format. Consistency of all components of the XT bundle is continuously monitored using extensive unit and integration tests (see [83]). The XT documentation is organized and maintained with Wiki technology and contains usage information of the individual tools as well as *HowTo's* which describe how these tools can be combined to perform specific transformation tasks.XT is completely component-based, which means that it promotes extensive reuse (see Figure 3.1 on page 47), that it can be extended with new components supporting the ATERMS exchange format, and that existing components can be replaced at any time. Language-centered software engineering by reusing XT components and developing additional ones is demonstrated in Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages", and Chapter 6, "Source Tree Composition".

3.5 Experience

In this section we describe some of our experiences with XT in various program transformation projects. For each project we indicate which program transformation scenarios needed to be addressed, and which XT constituents were (re)used.

Compilation of Tiger programs   A compiler for Appel's Tiger language [2] was developed as an exercise in compilation by transformation for a course on *High-Performance Compilers* at Universiteit Utrecht [141]. The compiler translates Tiger programs to MIPS assembly code. This translation is achieved by a number of transformations. Tiger abstract syntax is translated to an intermediate representation. The intermediate representation is canonicalized by a normalizing transformation. Canonicalized IR is translated to a MIPS program by instruction selection. Finally, register allocation optimizes register use by mapping temporary registers to actual machine registers. Optimizing transformations can be plugged in at various stages of compilation. These transformations have been implemented in Stratego. In addition, the compiler consists of a parser generated from an SDF grammar, a type-checker implemented in Stratego and a pretty-printer for Tiger built with GPP.

Warm fusion of functional programs   An implementation of a transformation system for a subset of HASKELL incorporating the warm fusion algorithm was undertaken as a case study in program transformation with rewriting strategies [75]. The warm fusion algorithm rephrases explicitly recursive functions as functions defined using catamorphisms to enable elimination of intermediate data structures (deforestation) of lazy functional programs. By inlining functions rephrased in this manner, compositions of functions can be fused. The bodies of all function definitions are simplified using standard reduction rules for functional programs.

The transformation system consists of a parser, a normalization phase to eliminate syntactic sugar, a type-checker, the warm fusion transformation itself and a pretty-printer. The grammar for HASKELL98 has been semi-automatically reengineered from a YACC grammar using the `yacc2sdf` tool. A pretty-printer for HASKELL was built using GPP. The transformations have been implemented in Stratego and make extensive use of the generic algorithms in the Stratego library, in particular those for substitution, free variable extraction and bound variable renaming.

Documentation generation for SDL   A documentation generator for the specification and description language SDL was built in collaboration with Lucent Technologies (see Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages"). AT&T's proprietary dialect of SDL was reengineered by automatically migrating an operational YACC definition to SDF. A suitable concrete syntax of SDL and a corresponding abstract syntax were constructed by applying several refactorings and optimizations to the generated SDF definition. Given the SDF definition, tools for documentation generation were constructed consisting of transformations for SDL code analysis and for visualization of SDL state transition graphs.

The SDL grammar was obtained from YACC using `yacc2sdf`, GPP was used for pretty-printing, and `sdfcons` was used for abstract syntax generation. Furthermore, the grammars used in addition to SDL where already available for reuse in the Grammar Base. All programming was performed with Stratego.

3.6   Measuring software reuse[2]

The software that we developed as part of the research covered in this thesis was developed with XT following the Language-Centered Software Engineering (LCSE) model presented in Chapter 2, "Grammars as Contracts".

To demonstrate the effectiveness of LCSE on software reuse, we present reuse statistics in each chapter that describes the development of a software *package* (i.e., a collection of components). These chapters contain a short paragraph "*components and reuse*", discussing component usage and software reuse

---

[2]This section is an extension to the originally published paper [84].

within that package. Components are considered to be executable programs (also called tools), fitting in the model for LCSE. Chapter 8 summarizes software reuse across all these packages in an overall picture.

Component usage Each paragraph "*components and reuse*" contains a figure displaying components that have been developed and components that are reused. For instance, Figure 3.1 on page 47 shows component usage for the XT package discussed in this chapter. Components are depicted as ellipses, packages, which are distribution units (i.e., collections of components that are collectively being developed and distributed) are denoted as boxes.

Light-grey boxes denote packages that have been developed during the research projects described in this thesis. Dark-grey boxes denote third-party packages. They originate from other projects in our group, such as the aterms package [28], or from other institutes such as the graphviz package [63]. Packages that have been discussed in a chapter are depicted as framed boxes.

Edges denote reuse relations. The source of an edge denotes the reusing component, the sink denotes the corresponding reused component. To reduce the number of edges, reuse relations over package boundaries are only displayed per package, not per component. Consequently, components do not have in or outgoing edges crossing package boundaries. The thickness of edges between packages corresponds to the number of reused components. The thicker an edge, the more components from a target package are reused.

Reuse levels Each paragraph also contains a table displaying information about component sizes and measurements of software reuse. Component sizes are indicated in lines of code (LOC). LOC as metric has known deficiencies but is also a reliable indicator of software size [118, 134] and is recommended by the Software Productivity Consortium [48]. Therefore, we will adopt this metric to measure software size. A discussion about how we calculate LOC follows shortly.

To quantify the amount of software reuse, we follow the de facto standard in industry and measure a component's *reuse level* as percentage:

$$\text{reuse level} = \frac{\text{Reused Software}}{\text{Total Software}} * 100\%$$

An alternative, equivalent expression is *reuse ratio* [133], but we will keep up with the terminology used in [118].

To make our measurements meaningful, we need to define exactly which source lines we count and which not. Furthermore, to be able to compare our measurements with reuse levels of other groups and institutes, we need to conform to a standard counting model. To that end, we will use the notion of *Reused Source Instructions* (RSI) and the reuse percent metric *Reuse%* [119], which is defined as:

$$Reuse\% = \frac{RSI}{Total\ Statements} * 100\%$$

Thus, a *Reuse%* of 100% corresponds to programs consisting of solely reused source code, while a *Reuse%* of 0% corresponds to programs that have been written completely from scratch.

RSI corresponds to software that complies to a number of rules regarding reuse.[3] Its purpose is to provide a standard definition of what to measure as reuse. Below we briefly enumerate some of these rules. The complete definition of RSI is discussed in [118].

1. RSI considers black-box reuse. White-box reuse in terms of modified components is not counted.

2. RSI makes no distinction between different programming languages. As a consequence, reusing a line of code in one language counts the same as reusing a line of code in any other language.

3. Each component is counted only once. Only the first use of a component therefore counts as reuse.

4. RSI measures complete components, even when a component's functionality is only partly used.

5. Unreachable (or dead) code in a reused component is counted as reuse.

6. Transitive reuse through component invocation is counted.

To measure the reuse level of the software that is discussed in this thesis, we will use the *Reuse%* metric, based on a slightly changed definition of RSI. Below we indicate how we deviate from the definition in [118]:

- LOC is influenced by the way programs are visually formatted. When software is reused from different institutes developed by a wide range of developers, differences in program layout must be ignored in order to calculate *Reuse%* accurately. Our measuring therefore involves pretty-printing in order to measure equally formatted programs.

- Source modules may contain comments, which affect the LOC of a component. The implications of comments in LOC comparisons is not discussed in [118]. To be independent from comments, we remove them prior to our measurements.

- LOC counts between different programming languages cannot easily be compared. Therefore, we measure software reuse for a single programming language only.

- We only count code that is accessible from a component's call graph. Code that is unreachable from the call graph is considered dead, and automatically removed prior to our measurements.

---

[3]Although its name might suggest that RSI is based on counting individual source instructions, it is based on line counting.

Components that invoke (execute) other components, are called composite components. Computation of *Reuse%* for a composite component therefore involves the component itself, as well as all components that it transitively invokes. Since, according to rule 4, we count complete components and not just the functionality that is accessed, the outcome of the computation soon becomes too optimistic. General usable components, which are often very flexible and contain much more functionality than is usually needed, make this miscalculation even worse. To make our measurements more realistic, we provide two reuse levels. An optimistic (transitive) one, which counts RSI transitively for a component and for all components that it invokes, and a pessimistic (non-transitive) one, which counts RSI only for a single component.

Table 3.1 depicts reuse levels for the components of the XT package. The first column shows the list of components that are part of the package. Columns 2–4 depict pessimistic measurements, corresponding to non-transitive reuse (i.e., reuse within a single component). Columns 5 and 6 show optimistic values, corresponding to transitive software reuse. The last row contains accumulated reuse levels for all components together. If a component's RSI equals 0 (such as for `tohtml-sdf`), then the component is completely written from scratch without reusing a single line of code. This might, for example, be the case for tiny "glue" tools, which only invoke other components. It typically results in a high transitive, rather than a high non-transitive *Reuse%*. If a component's non-transitive reuse equals transitive reuse (as is the case for the `atermdiff` component), then the component is completely self-contained and does not invoke any other components.

We only measure reuse levels for Stratego [142]. This programming language is used for the implementation of most components discussed in this thesis. Reuse of components implemented in other languages is not counted, although they are frequently used. These third-party components together amount for more than 200,000 LOC and would completely obfuscate the reuse levels of the software discussed in this thesis. As a consequence, the statistics shown do not give a complete picture of actual software reuse. In reality, software reuse is better than the tables suggest.

The numbers in the tables are obtained by automatic source code analysis. Per component the total number of LOC and the number of RSI are calculated. Line counting is based on equally formatted and normalized programs, discarding code that is unreachable from a component's call graph. Normalization reduces the number of used language constructs by removing syntactic sugar. Pretty-printing produces equally formatted modules by ignoring comments and personal format conventions. Thus, normalization and pretty-printing improve comparability of modules, which might have been implemented by different persons in completely different styles. This makes line counting appropriate as reuse statistic. Due to normalization and pretty-printing, LOC explodes 148% on average. Thus, due to this *explosion factor*, component sizes are, on average, 1.48 times smaller then the tables indicate. The explosion factor does not

influence a component's *Reuse%*.

The non-transitive number of LOC in the second column is determined as follows. First, all Stratego source modules that are used by the implementation of the component are collected and parsed to obtain an abstract syntax tree (AST). Then, parts of the Stratego compiler are used to perform normalization steps and to remove dead-code. The result is an AST of the normalized Stratego program containing *only* used code. This AST is then transformed to plain text using the generic pretty-printer GPP discussed in Chapter 4, "Pretty-Printing for Software Reengineering". Finally, empty lines are removed from the resulting program and the number of lines is counted.

The non-transitive number of RSI in the third column is determined as follows. First, the set of component-specific Stratego modules is determined. These are the modules that are used by a component and which are located in the source directory of that component. They are parsed to obtain an AST of component-specific Stratego code. Next, the number of component-specific LOC is determined by normalizing and pretty-printing the AST as described above. By subtracting this number from the total number of LOC, the number of RSI is obtained.

The transitive number of LOC in the fifth column corresponds to the total number of LOC of the component. It is determined by first computing the set of components that is transitively invoked by a component, and then accumulating the LOC of each of them including the component itself. The transitive *Reuse%* in the last column is computed as:

$$\frac{\text{LOC}_{\text{transitive}} - (\text{LOC}_{\text{non-transitive}} - \text{RSI}_{\text{non-transitive}})}{\text{LOC}_{\text{transitive}}}$$

For the code analysis we used components from the Stratego compiler, the pretty-printer GPP, and some newly developed components. The analysis is therefore itself a language tool which demonstrates software reuse in the domain of language tooling. Thus, its development is an example of LCSE as proposed in Chapter 2, "Grammars as Contracts".

3.7   Concluding remarks

Availability   XT and all its constituent components are distributed as open source under the GNU General Public License [60], and anyone is allowed to use, modify, and redistribute them.[4] The distribution makes use of `autobundle`, `autoconf`, and `automake`, which make installation a nearly trivial job by merging the build and configuration processing of the individual components (see Chapter 6, "Source Tree Composition"). XT is known to install and run successfully on various platforms, among which SUN-Solaris, BSD-Unix, Linux, and Windows.

---

[4]See Appendix A for information about the availability of the XT bundle.
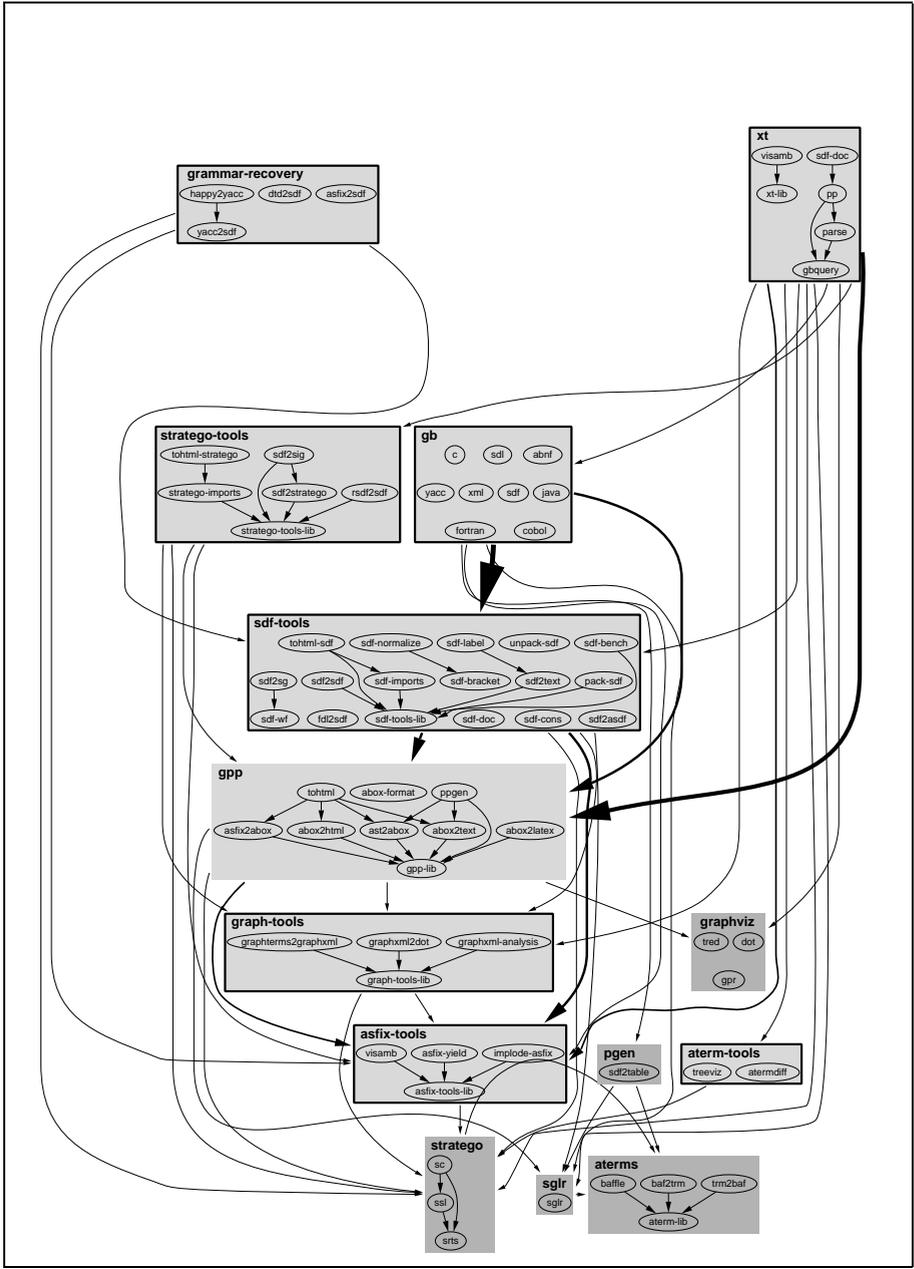
Figure 3.1 Components used for the implementation of XT.

Comparison to other frameworks   XT shares its bundling infrastructure and the SDF and ATERMS packages with a peer bundle: the ASF + SDF Meta-Envir-

| Component | Non-transitive reuse | | | Transitive reuse | |
	LOC	RSI	Reuse%	LOC	Reuse%
atermdiff	1,221	1,065	87%	1,221	87%
yacc2sdf	2,427	1,905	78%	2,427	78%
GraphXML2dot	1,054	863	81%	1,054	81%
sdf2asdf	1,294	929	71%	1,294	71%
sdf-label	1,652	1,500	90%	1,652	90%
sdf2sg	1,606	1,286	80%	1,606	80%
tohtml-sdf	7	0	0%	12,405	99%
sdf-doc	1,001	897	89%	1,001	89%
pack-sdf	1,504	1,352	89%	1,504	89%
sdf-bracket	868	775	89%	868	89%
sdf2text	47	0	0%	8,312	99%
sdf2sdf	672	567	84%	14,955	99%
sdf-wf	1,350	1,089	80%	1,350	80%
sdf-imports	1,410	1,107	78%	1,410	78%
sdf2stratego	2,300	1,319	57%	2,300	57%
implode-asfix	1,708	693	40%	1,708	40%
pp	76	0	0%	11,724	99%
parse	53	0	0%	1,873	97%
gbquery	112	0	0%	112	0%
Totals:	**20,362**	**15,347**	**75%**	**68,776**	**92%**

Table 3.1 Reuse table for a subset of the 70+ components from the XT bundle. The table shows that for these components, a total of 5,015 *new* lines of code had to be written.

onment [27]. This bundle integrates these packages with a compiler and interpreter for the ASF programming language, a structure editor, a GUI, and other components into an *interactive development environment* for language definitions and tools. By contrast, XT supports multiple programming languages, and offers an extendible set of components that can combined in various ways.

Many tools and frameworks for program transformation, or for some of its sub-scenarios, already exist. Among these are attribute grammar systems (e.g. Elegant [3]), algebraic rewriting systems (e.g., ASF+SDF Meta-Environment [27], ELAN [20]), and object-oriented systems (e.g., the Smalltalk refactoring browser [123] and OPENC++ [42]). See [143] for a more complete overview of transformation frameworks. Generally, these systems are closed in the sense that they provide a fixed set of tightly-coupled components (such as parser, pretty-printer, and transformation language), they have no support for exchange or interoperation with other (competing) systems, and they are

biased towards a single programming language.

XT does not attempt to compete with these systems by providing yet another closed transformation tool. Instead it reuses components from existing systems, and demonstrates how they can be used in a completely open, extendible framework. Different constellations of transformation tool bundles can be obtained by adding new components to XT, which can supplement or replace the current ones. Also, one can use XT as a basis for the creation of specific (possibly closed) transformation frameworks for particular application areas, or for particular source and target languages (see for instance CODE-BOOST, a framework for C++ program transformation [5]).

Components and reuse Figure 3.1 displays the packages bundled with XT and their constituent components (see Section 3.6 on page 42 for more information about component diagrams). XT bundles 14 packages containing 73 tool components as well as a collection of 36 grammars (only 9 grammars are depicted to prevent clutter). The collection of packages includes 4 third-party packages, containing 8 third-party components, as well as the gpp package which will be discussed in the next chapter. The picture is not complete because, in order to prevent cluttering of the picture, we only included the most important XT components. Moreover, since XT is evolving rapidly, the number of packages and components is still growing.

Table 3.1 depicts component sizes and reuse levels of a subset of the XT components. This table shows that these XT components consist of more than 20,300 lines of code, of which more than 15,300 lines are reused. This yields a reuse level between 75% and 92%. All XT components of the light-grey packages together consist of 65,719 lines of code, of which 52,560 lines are reused (see the summary of software reuse in Chapter 8 on page 139). This yields a reuse level for XT between 80% and 91%. Section 3.6, "Measuring software reuse", justifies these numbers and describes how they are obtained by analyzing component implementations. In Chapter 8, "Conclusions", we will compare these figures with reuse levels of other projects discussed in this thesis.

Pretty-Printing for Software Reengineering

Pretty-printing forms an integral part of Language-Centered Software Engineering (LCSE). To facilitate reuse, generic (i.e., language-independent) and customizable pretty-print technology is needed. In this chapter we discuss such pretty-print technology in the context of software reengineering.

A typical application domain of LCSE is software reengineering. Software reengineering puts strong requirements on pretty-print technology. These include layout and comment preservation, as well as customizable format definitions. From a maintenance perspective, software reengineering requires reusability of format engines and of format definitions.

In this chapter we present the Generic Pretty-Printer GPP *and discuss the pretty-print techniques that it uses to fulfill the requirements for software reengineering.* GPP *forms a generally reusable pretty-print component in our language-centered architecture and is part of the* XT *bundle discussed in the previous chapter. Applications, such as* COBOL *reengineering and* SDL *documentation generation (which will be discussed in the next chapter) show that our pretty-print techniques are feasible and successful. The work presented in this chapter was published earlier as [80].*

4.1 Introduction

Software reengineering is concerned with changing and repairing existing software systems. It is often language-dependent and customer-specific.

For instance, Dutch banks have to standardize their bank account numbers

before the second quarter of 2004 [105]. To that end, a restructuring reengineering [43] might be implemented for a particular Dutch bank to reengineer his COBOL-85 dialect, by changing account numbers from 9 to 10 digits while preserving his specific coding conventions. Although the reengineering itself is of general use for all Dutch banks, this specific implementation is hard to reuse.

When a reengineering company wants to develop such reengineerings for different customers and different language dialects (for instance to support the bank account number reengineering for some other of the 300 existing COBOL dialects [95]), problematic reuse may easily lead to a significant maintenance effort. A reengineering company would therefore benefit when reuse of reengineerings could be improved, such that reengineerings for new customers or language dialects can be developed rapidly from existing ones and time to market can be decreased [114].

Developing reusable reengineerings requires advanced language technology to easily deal with multiple customers and language dialects. The literature contains many articles addressing flexible parsing and processing techniques. Flexible, reusable pretty-printing techniques are not very well addressed and are the subject of this chapter.

Pretty-printing in the area of software reengineering serves two purposes. Firstly, for *automatic* software reengineering pretty-printing is used for *source (re-) generation*, to transform the abstract representation of a reengineered program back to human readable textual form. Usually, a pretty-printer is then the last phase in a reengineering pipeline. Programs are first parsed, then reengineered (for instance by transformation), and finally pretty-printed.

Secondly, for *semi-automatic* (or manual) software reengineering pretty-printing is used for *documentation generation* [57]. In this case, the reengineering process requires user intervention and documentation generation is used to make programs easily accessible. To that end, a pretty-printer is used to format programs nicely and can be combined with additional program understanding techniques to enable analysis and inspection of programs to determine where and how reengineering steps are required. Examples are web-site generation and document generation.

The area of software reengineering introduces challenging functional requirements for pretty-printing:

- To support source generation and documentation generation, a pretty-printer should be able to produce multiple output formats. We define formattings independently of such output formats in *pretty-print rules* [106].

- The modifications made during software reengineering to the original source text should be minimal [136]. Only the parts that need repairing should be modified. This implies that comments and existing layout should be preserved. To that end, we propose *conservative* pretty-printing.

- Customer-specific format conventions should be respected. To yield a program text that looks like the original one, a pretty-printer is required that produces a customer-specific formatting. We propose *customizable* pretty-printing to meet such specific conventions.

In order to cost-effectively develop pretty-printers for various programming languages, tailored towards different customer-specific format conventions, maintenance effort should be minimal. To reduce maintenance effort, reuse across different pretty-printers should be promoted. To that end, we formulate a number of technical requirements:

- A formatting definition should be reusable for a language, its dialects, and for defining customized formattings. We propose to group pretty-print rules in *modular pretty-print tables*, which allow a formatting to be defined as a composition of new and existing pretty-print tables.

- Adding support for a new language should be easy. We propose using generic formating engines, which can interpret pretty-print rules of arbitrary languages. Moreover, we simplify creating new formatting definitions with *pretty-print table generation*.

- Software reengineering can be applied to different representations of programs each requiring specific pretty-print techniques. We address two such representations (i.e., parse-trees and abstract syntax trees), and we propose to reduce maintenance cost by sharing formatting definitions between the corresponding formatting engines.

In contrast to the existing literature, which usually concentrates on a specific aspect of pretty-printing [79], this chapter strives to give a complete discussion of pretty-printing in the area of software reengineering. We start with a discussion of conceptual foundations of pretty-printing for software reengineering. Then we describe the generic pretty-print framework GPP. Finally, we cover existing reengineering projects conducted with GPP, including COBOL reengineering and SDL documentation generation.

4.2 Pretty-printing

Pretty-printing is concerned with transforming abstract representations of programs to human readable form. Pretty-printing is like unparsing, but additionally, is concerned with producing nicely (pretty) formatted output. The result of pretty-printing can be plain text which is directly human readable, or a document in some markup language such as HTML.

In this chapter we consider two types of abstract representations: (full) *parse trees* (see Figure 4.1) and *abstract syntax trees* (see Figure 4.2). Full parse trees (also called *concrete syntax trees*) contain all lexical information, including comments and ordinary layout.
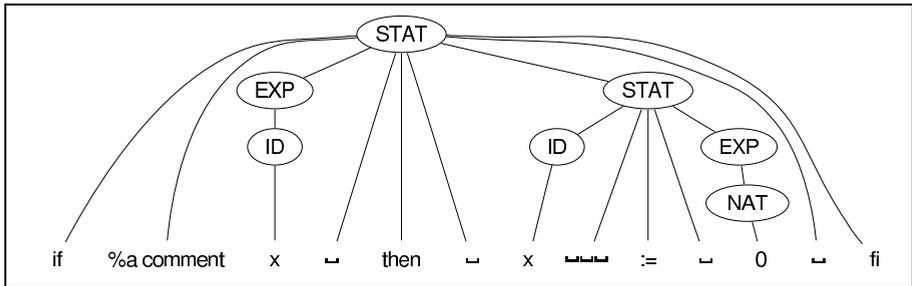
Figure 4.1 Example of a full parse tree containing layout and comments. Layout nodes are denoted by '␣', comments start with '%'.

A parse tree (PT) can be pretty-printed *progressively*, which means that *all* layout will be generated. For the PT of Figure 4.1 this implies that the comment and '␣' nodes are discarded and replaced by newly generated layout strings. Generation of layout can also be less progressive by preserving (some or all) of the existing layout nodes. We call this *conservative* pretty-printing.

An abstract syntax tree (AST) must always be pretty-printed progressively, since it does not contain layout nodes. An extra challenge of AST pretty-printing is that literals (i.e., the keywords of the program) should be reconstructed.

Pretty-printing consists of two phases [113]. First, the abstract representation is transformed to an intermediate format containing formatting instructions. Then, the formatting instructions are transformed to a desired output format.

The intermediate format that is obtained during the first phase of pretty-printing can be represented as a tree (see Figure 4.3). The nodes of this *format tree* correspond to format operators and denote how to layout the underlying leafs (for example, H for horizontal, and V for vertical formatting). This phase is thus basically a tree transformation in which an AST or PT is transformed to a format tree. During the second phase, the format tree is used to produce the corresponding layout in the desired output format.

We propose to define the transformation to a format tree as a mapping from language constructs (grammar productions) to corresponding format constructs. As we will see in Section 4.4, such mappings can be shared for transforming PTs and ASTs. This makes pretty-printing of both tree types, based on a single transformation definition, possible. We will also see in Section 4.4 how the construction of such language-specific mappings is simplified by generating them from corresponding grammars.
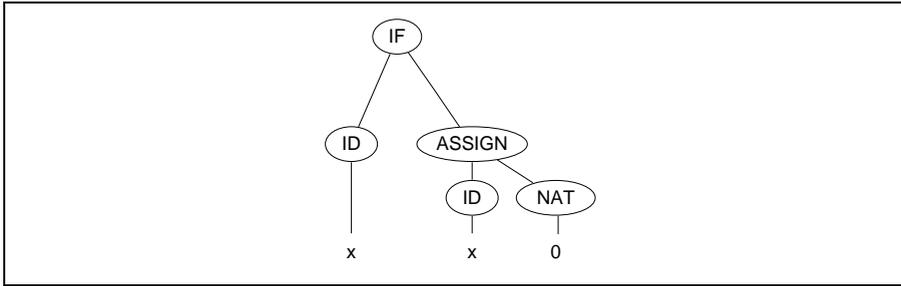
Figure 4.2 Example of an abstract syntax tree.

4.3 Pretty-printing for software reengineering

This section addresses requirements for pretty-printing in the context of software reengineering as well as corresponding solutions.

4.3.1 Multiple output formats

Pretty-printing for software reengineering serves two purposes: i) as back-end of an automated reengineering process; ii) as part of a documentation generator. In the first case a pretty-printer is used to transform a reengineered program to plain text, such that it can be further developed, compiled etc. In the latter case, it is used to produce a visually attractive representation of programs for *program understanding* purposes.

Both purposes demand for different output formats: plain text in case the pretty-printer serves as back-end, and a high-quality format (such as LaTeX, HTML, or PDF) for a documentation generator.

To limit maintenance cost of a pretty-printer due to code duplication we divide a pretty-printer in two separate components, a *format tree producer* and a *format tree consumer*. The first produces a language-specific formatting represented as format tree, while the latter transforms such a tree to an output format. This division makes a producer independent of the output format and a consumer independent of the input language.

A pretty-printer for input language i and output format o now consists of the composition:

$$pp_i^o = \textit{format tree producer}_i + \textit{format tree consumer}_o$$

By developing different format tree consumers, a format tree can be transformed to multiple output formats. In Section 4.4 we discuss the implementation of three such consumers which produce plain text, HTML, and LaTeX.

This architecture reduces maintenance effort because each format tree consumer can be reused as-is in all pretty-printers. Once the pretty-print rules for
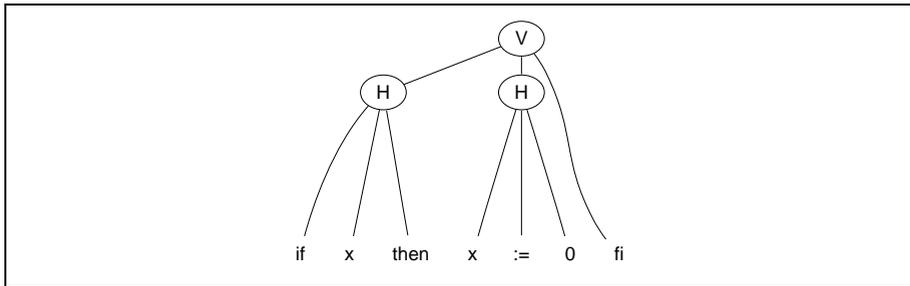
Figure 4.3 Format tree produced by progressive pretty-printing.

a language have been defined, programs in that language can be formatted in all available output formats without extra effort.

4.3.2   Layout preservation

An important function of layout is to improve the understandability of programs. Such layout is inserted by developers and does not always follow strict format conventions. It may contain slight adaptations, for instance to group certain lines of code together, or to make a statement fit nicely on a single line.

With standard (progressive) pretty-print techniques such formattings will disappear. Consequently, layout occurring in program fragments that are not even touched by the actual program reengineering will not survive.

Clearly, for serious software reengineering, it is essential that the formatting of unaffected program parts *will* be preserved [136]. Only the affected parts should be formatted automatically using a customized pretty-printer. Therefore the use of *conservative* pretty-printing is inevitable.

Conservative pretty-printing produces a format tree which may contain original layout nodes (such as the node '␣␣' in Figure 4.1). Conservative pretty-printing therefore only works on full parse trees because ASTs, in general, do not contain layout nodes.

Conservative pretty-printing operates on PTs in which the layout nodes that should be preserved are marked. We defined an algorithm for conservative pretty-printing that consists of two steps. First, a PT is mapped to a format tree using progressive pretty-printing. Second, the nodes that were marked in the original PT are inserted in the format tree. To combine these layout nodes with non-layout nodes, we introduce a special empty format operator ϵ. This operator joins its sub-trees without producing any layout. For each layout node that needs to be preserved, the insertion process is defined as follows:

1. The terminal symbol occurring left to a layout node is removed from the format tree (e.g., the second 'x' in Figure 4.3 in case the layout string '␣␣' between the symbols 'x' and ':=' in Figure 4.1 has to be preserved).
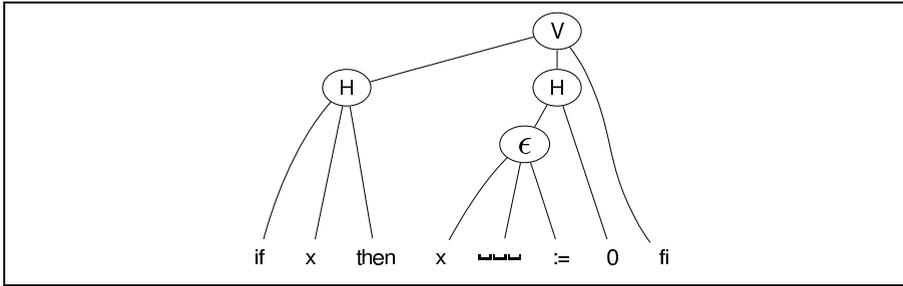
Figure 4.4 Format tree in which existing layout (␣␣) is preserved.

2. The format tree right to that terminal symbol is determined and removed (the node ':=' in Figure 4.3).

3. The terminal symbol ('x'), the layout node ('␣'), and the format tree (':=') are then inserted into a new sub-tree which has the empty format operator ϵ as root.

4. This tree is inserted in the original format tree, at the location of the terminal symbol that was removed at step 1.

Applying these steps to the format tree of Figure 4.3 yields the tree as depicted in Figure 4.4.

To transform a format tree with layout nodes to text, we use a format function f that operates on format expressions and produces text. It is defined as follows:

$$
\begin{aligned}
f(\epsilon(t_1, \ldots, t_n)) &= f(t_1) \cdot \ldots \cdot f(t_n) \\
f(\phi(t_1, \ldots, t_n)) &= f(t_1) \cdot l_\phi \cdot f(t_2) \cdot \ldots \cdot l_\phi \cdot f(t_n) \\
&\quad \text{for each format operator } \phi \\
f(s) &= s \text{ for each non-terminal symbol } s \\
f(w) &= w \text{ for each layout string } w
\end{aligned}
$$

The operator '·' denotes string concatenation, l_ϕ denotes the layout string as generated by the format operator ϕ. This definition states that sub-trees of ϵ are only concatenated, while sub-trees of other operators are also separated by layout strings.

For example, when we define $l_H = $ ␣ and $l_V = \backslash n$, then we can translate the format tree of Figure 4.4 to text using the following derivation:

$$
\begin{aligned}
f(V(H(\mathsf{if}, \mathsf{x}, \mathsf{then}), H(\epsilon(\mathsf{x}, \text{␣␣␣}, \mathsf{:=}), \mathsf{0}), \mathsf{fi})) &= \\
f(H(\mathsf{if}, \mathsf{x}, \mathsf{then})) \cdot l_V \cdot f(H(\epsilon(\mathsf{x}, \text{␣␣␣}, \mathsf{:=}), \mathsf{0})) \cdot l_V \cdot \mathsf{fi} &=
\end{aligned}
$$
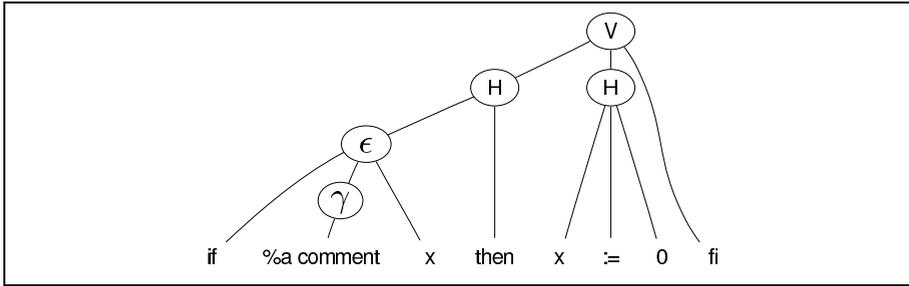
Figure 4.5 Format tree with preserved comments.

$$\text{if} \cdot l_H \cdot \text{x} \cdot l_H \cdot \text{then} \cdot l_V \cdot f(\epsilon(\text{x}, \text{\textvisiblespace\textvisiblespace\textvisiblespace}, :=)) \cdot l_H \cdot 0 \cdot l_V \cdot \text{fi} \quad =$$
$$\text{if} \cdot l_H \cdot \text{x} \cdot l_H \cdot \text{then} \cdot l_V \cdot \text{x} \cdot \text{\textvisiblespace\textvisiblespace\textvisiblespace} \cdot := \cdot l_H \cdot 0 \cdot l_V \cdot \text{fi} \quad =$$
$$\text{if\textvisiblespace x\textvisiblespace then} \backslash n \text{x\textvisiblespace\textvisiblespace\textvisiblespace} := \text{\textvisiblespace} 0 \backslash n \text{fi}$$

Since the ϵ operator produces no layout, the string that separates the two adjacent terminal symbols 'x' and ':=' is exactly the layout string that had to be preserved. All other layout strings are generated according to the (customer-specific) pretty-print rules.

4.3.3   Comment preservation

Like layout, comments also serve to improve the understandability of programs. Such information, which might include important descriptions and instructions to developers, should in all cases be preserved. Since an ordinary progressive pretty-printer would destroy this information, a comment preserving pretty-printer is required for software reengineering.

Comment preservation is similar to layout preservation and is also only defined on full parse trees. The ϵ operator is used to insert a comment into a format tree (like for inserting ordinary white space). In addition, we introduce a new format operator γ to mark comments in format trees. This operator serves documentation generation and *literate programming* [89]. It allows comments to be formatted explicitly in non-text formats (for instance in a separate font).

The format tree that is obtained from Figure 4.3 by inserting '%a comment' is depicted in Figure 4.5.

4.3.4   Customizability

When performing automatic software reengineering for a customer, the resulting programs should have a similar formatting as the original ones. When different customers are served, this requires the availability of several format engines, each producing the formatting of a particular customer. Developing

each of them from scratch is a lot of work and easily leads to undesired maintenance effort. Instead, reusing existing pretty-print engines and customizing their behavior is preferable.

To make pretty-printers easily customizable, we advocate pretty-printing using *pretty-print rules* [106]. Pretty-print rules are mappings from language constructs to formatting constructs. Each mapping defines how a language construct should be formatted. Pretty-print rules are defined declaratively and interpreted by a formatting engine. Constructing a format tree from a PT or AST now consists of a tree transformation in which the exact transformation is defined by a set of pretty-print rules. Customization is achieved by supplying different rules to a formatting engine.

By using this interpreted approach, the same formatting engine can be used for every language and all customers. Only pretty-print rules have to be defined to develop a pretty-printer for a language. Furthermore, existing rules can be redefined to customize a formatting definition.

4.3.5 Modularity

Software reengineering requires pretty-print technology that makes dealing with language dialects, embedded languages, and customer-specific formattings easy.

To facilitate this, development and maintenance time should be decreased by allowing new pretty-printers to be constructed from existing ones. In Section 4.3.4, we already pointed out how this can be achieved by separating pretty-print rules and formatting engines. Pretty-print engines can be reused for all different customers, only pretty-print rules have to be defined for each of them.

However, only a small portion of an existing set of pretty-print rules needs to be changed usually, when adding support for a new customer-specific format or language dialect. Pretty-printer construction would therefore be further simplified when the unchanged pretty-print rules could also be reused. With modular pretty-print tables this is achieved.

We therefore group pretty-print rules in *pretty-print tables*. By prioritizing each table, pretty-print rules in a table with higher priority will override the rules in tables of lower priority. The set of pretty-print rules ρ obtained by combining two tables t_1 and t_2 (where t_1 has highest priority), is defined as:

$$\rho = t_1 \cup (t_2 \setminus t_1)$$

Definition of formattings with modular pretty-print tables works as follows:

Language dialects Pretty-print rules corresponding to new or affected language constructs are defined in a new set of pretty-print tables T_{new}. The complete formatting definition is obtained by merging T_{new} with the pretty-print tables of the existing (base) language. Thus, only the pretty-print rules in T_{new} have to be defined, the rest can be reused.
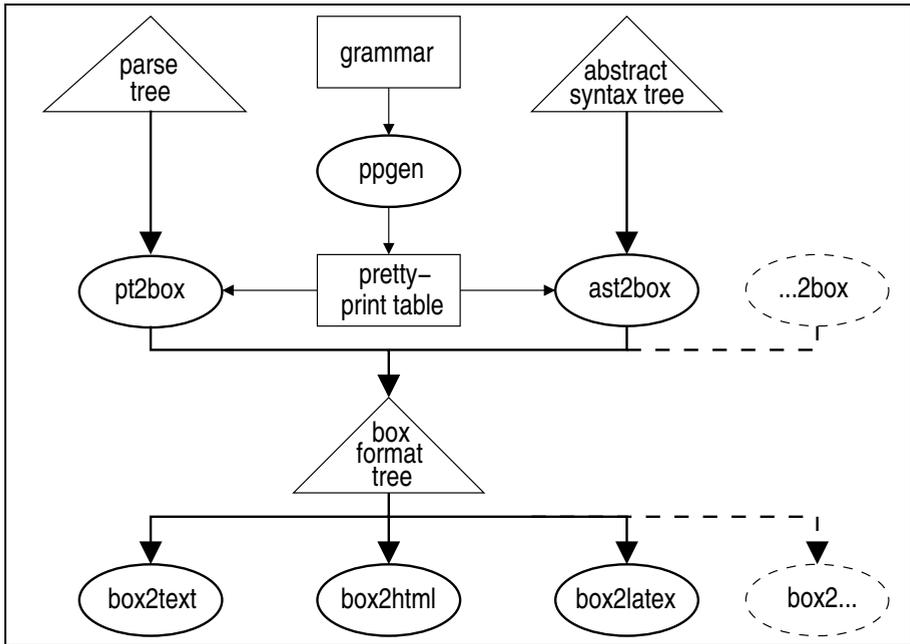
Figure 4.6 Architecture of the generic pretty-printer GPP. Ellipses denote GPP components, boxes and triangles denote data.

Embedded languages A formatting definition for a language L which embeds L_e is obtained by combining the format rules of L with the format rules of L_e. For instance, suppose you already have pretty-print rules for the language COBOL and SQL, then building a pretty-printer for COBOL with embedded SQL, amounts to combining both sets of pretty-print rules.

Customer specificity A customer-specific formatting definition for a language L can be defined by combining customer-specific pretty-print rules and standard pretty-print rules. Only the customer-specific rules have to be defined which override the reused default rules.

An arbitrary number of pretty-print tables can be merged this way, which makes extensive reuse of pretty-print rules possible. For instance, to define a customer-specific formatting definition for an embedded language dialect.

4.4 GPP: a generic pretty-printer

GPP is a generic pretty-printer that implements the ideas and techniques discussed in Section 4.2 and 4.3.[1] GPP's architecture is shown in Figure 4.6. It

---

[1] See Appendix A for information about the availability of GPP.

```
context-free syntax
  "if" EXP "then" STAT "fi" → STAT {cons("IF")}
  ID ":=" EXP              → STAT {cons("ASSIGN")}
  ID                       → EXP  {cons("ID")}
  NAT-CON                  → EXP  {cons("NAT")}
  STR-CON                  → EXP  {cons("STR")}
```

Figure 4.7 Grammar productions in the Syntax Definition Formalism SDF, corresponding to the PT and AST of Figure 4.1 and 4.2.[2]

consists of the pretty-print table generator `ppgen`, two format tree producers `pt2box` and `ast2box` (see Section 4.4.2), and the format tree consumers `box2text`, `box2html`, and `box2latex` (see Section 4.4.3). Format tree producers and consumers are implemented as separate components which exchange format trees represented in the markup language BOX. This architecture can easily be extended by additional tools that produce or consume BOX format trees (see Section 4.5).

4.4.1 Format definition

The first phase of pretty-printing consists of transforming a PT or AST to a format tree. This section discusses how the transformation can be defined as mapping from language productions in SDF to format constructs in BOX.

Syntax definition   We use the Syntax Definition Formalism SDF [68, 137] to define language productions (see Figure 4.7 for some examples). SDF allows modular, purely declarative syntax definition. In combination with generalized LR (GLR) parser generation, the full class of context-free grammars is supported.

SDF allows concise syntax definitions with syntax operators in arbitrary nested productions. For instance, using the '()' and '?' operators which denote sequences and optionals, respectively, the if-construct of Figure 4.7 can be extended with an optional else-branch as depicted in Figure 4.8.

Together with concrete syntax, corresponding abstract syntax can also be defined in SDF. This is achieved by defining the constructor names of the abstract syntax as annotations to the concrete syntax productions (the cons attributes in Figure 4.7 and 4.8). These define the node names of abstract syntax trees. Chapter 2, "Grammars as Contracts", discusses generation of abstract syntax from concrete syntax in more detail.

---

[2]Productions in SDF are reversed with respect to formalisms like BNF: on the right hand side of the arrow is the non-terminal symbol that is produced by the symbols on the left-hand side of the arrow.

```
context-free syntax
  "if" EXP "then" STAT ( "else" STAT )? "fi" → STAT {cons("IF")}
```

Figure 4.8 Extending the if-construct of Figure 4.7 with an optional else-branch using the SDF syntax operators '()' and '?'.
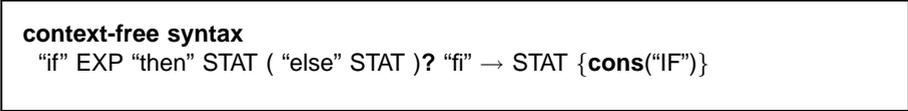
As we will see shortly, constructor names also serve to identify productions and to select proper pretty-print rules.

Format expressions We use the language BOX [33, 79] for defining formattings. BOX is a markup language which allows formatting definitions to be expressed as compositions of boxes. Box composition is accomplished using *box operators* (Table 4.1 lists available operators).

The language distinguishes *positional* and *non-positional* operators. Positional operators are further divided in *conditional* and *non-conditional* operators. Examples of non-conditional, positional operators are the H and V operators, which format their sub-boxes horizontally and vertically, respectively:

$$\mathsf{H}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,]\ \ =\ \ \boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}$$

$$\mathsf{V}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,]\ \ =\ \ \begin{array}{c}\boxed{B_1}\\\boxed{B_2}\\\boxed{B_3}\end{array}$$

Conditional operators take the available line width into account. An example is the ALT operator:

$$\mathsf{ALT}\,[\,\boxed{B_1}\,\boxed{B_2}\,]\ =\ \ \boxed{B_1}\ \ or\ \ \boxed{B_2}$$

It either formats its first sub-box if sufficient width is available, or its second otherwise.

The exact formatting of positional operators can be controlled using *space options*. For example, to control the amount of horizontal space between boxes, the H operator supports the hs space option:

$$\mathsf{H_{hs=2}}\,[\,\boxed{B_1}\,\boxed{B_2}\,\boxed{B_3}\,]\ =\ \ \boxed{B_1}\,\text{␣␣}\,\boxed{B_2}\,\text{␣␣}\,\boxed{B_3}$$

The non-positional operators of the BOX language are used for cross referencing and for specifying text attributes (such as font and color). They are also used to structure text, for instance by marking parts as comment text, as variable, or as keyword.

BOX does not have an explicit empty format operator ϵ, which is needed for conservative pretty-printing (see Section 4.3.2). However, this operator can

	Operator	Options	Description
Positional	H	hs	Horizontal formatting of sub-boxes
	V	vs, is	Vertical formatting of sub-boxes
	HV	hs, vs, is	Horizontal and vertical formatting of sub-boxes, taking line width into account
	A	hs, vs	Formatting of sub-boxes in a tabular
	R		Grouping of rows in a tabular
	ALT		Conditional formatting depending on available line width
Non-positional	F		Operator to specify fonts and font attributes
	KW		Font operator to format keywords
	VAR		Font operator to format variables
	NUM		Font operator to format numbers
	MATH		Font operator to format mathematical symbols
	LBL		Operator used to define a label for a box
	REF		Operator to refer to a labeled box
	C		Operator to represent lines of comments

Table 4.1 Positional and non-positional BOX operators, together with supported space options (hs defines horizontal layout between boxes, vs defines vertical layout between boxes, and is defines left indentation).

be mimicked using the H operator as $H_{hs=0}$ [...]. The comment operator γ used for comment preservation (see Section 4.3.3) is represented using the C operator in BOX.

Pretty-print rules   We can now define pretty-print rules as mappings from SDF productions to BOX expressions, and pretty-print tables as comma separated lists of pretty-print rules (see Figure 4.9).

BOX expressions in pretty-print tables contain numbered place holders (_1 and _2 in Figure 4.9) which correspond to non-terminal symbols in SDF productions. During pretty-printing, place holders are replaced by BOX-expressions that are generated for these non-terminal symbols.

```
IF        -- V[ H[ KW[ "if" ] _1 KW[ "then" ] ] _2 KW[ "fi" ] ],
ASSIGN -- H[ _1 KW[ ":=" ] _2 ],
ID        -- _1,
NAT      -- _1,
STR      -- _1
```

Figure 4.9   A pretty-print table for the grammar of Figure 4.7.

```
IF              -- V[ H[ KW[ "if" ] _1 KW[ "then" ] ] _2 _3 KW[ "fi" ] ],
IF.3:opt        -- _1,
IF.3:opt.1:seq  -- KW[ "else" ] _1
```

Figure 4.10 Pretty-print table for the nested if-construct of Figure 4.8.

Constructor annotations of SDF productions serve as keys in pretty-print tables. Since they are contained in the parse tree format that we use (see Section 4.4.2) and (as node names) in ASTs (see Figure 4.2), they can be used to format both tree types.

The pretty-print table of Figure 4.9 only contains pretty-print entries for flat (non-nested) SDF productions. To enable the definition of a formatting for an arbitrary nested SDF production, a separate pretty-print rule can be defined for each nested symbol in a production. Such pretty-print rules are identified by the path from the result sort of the production to the nested symbol.

For example, Figure 4.10 contains pretty-print entries for the nested SDF production of Figure 4.8. The first rule has only the constructor name (IF) as key and corresponds to the top-level sequence of symbols of the SDF production (i.e., the sequence of children of the root node of the tree depicted in Figure 4.11). The remaining pretty-print rules correspond to the nested symbols. For each path from the root node in Figure 4.11 to a nested symbol (denoted as grey ellipse), a pretty-print rule is defined.

All path elements contain indexes which are obtained by numbering the non-terminal symbols contained in nested symbols. Symbol names are required as part of path elements for pretty-printing ASTs. This is further discussed in Section 4.4.2.

Pretty-print rule generation   Writing pretty-print tables can be a time consuming and error-prone process, even for small languages. To simplify pretty-print table construction, we implemented a pretty-print table generator (the component ppgen in Figure 4.6). Given a syntax definition in SDF, this generator produces pretty-print rules for all SDF productions and for all paths to nested symbols.

Literals in SDF productions are recognized as keywords and formatted with the KW operator. Several heuristics are used to make a rough estimation about vertical and horizontal formatting. This is achieved by recognizing several structures of SDF productions and generating compositions of H and V boxes accordingly. For most productions however, no explicit formatting definition is generated. That is, a pretty-print rule is generated without positional BOX-operators (such as the last two pretty-print rules in Figure 4.10). This makes generated tables easy to understand and to adapt.

With the generator, a pretty-printer for a new language can be obtained for free. Although the formatting definition thus obtained is minimal, it is di-
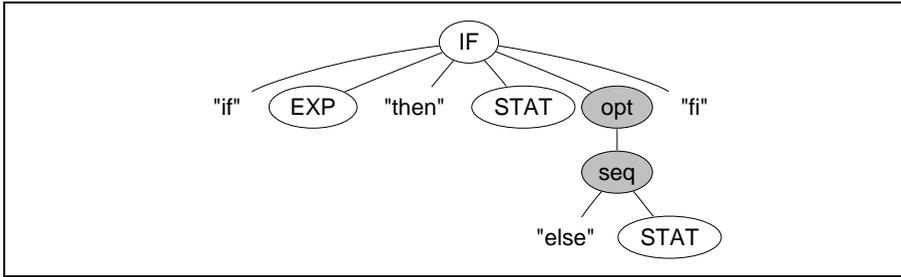
Figure 4.11 Graphical structure of the nested SDF production of Figure 4.8. Grey ellipses denote nested SDF symbols.

rectly usable. To improve the layout, we can benefit from the customizability of pretty-print tables and incrementally redefine the formatting definition. Only pretty-print rules that do not satisfy have to be redefined. Customized pretty-print rules can be grouped in separate tables such that regeneration of tables after a language change does not destroy the customized rules.

4.4.2 Format tree producers

This section discusses `pt2box` and `ast2box` which generate format trees from PTs and ASTs, respectively.

Formatting parse trees Formatting PTs is implemented in `pt2box` which supports conservative and comment preserving pretty-printing. It operates on a universal PT format, called AsFix (see Chapter 2, "Grammars as Contracts"). This format can represent PTs for arbitrary context-free languages and contains all lexical information including layout and comments. In addition, constructor symbols are also available because each node in an AsFix PT contains the complete grammar production that produced the node (see Figure 2.4 on page 23 for an example of an AsFix PT).

The format tree producer `pt2box` first collects all layout nodes from a PT. Then it constructs a BOX-expression during a bottom-up traversal of the PT. Finally, it inserts some of the collected layout nodes in the BOX-term. Layout is inserted as follows: comments are always inserted; original non-comment layout is only inserted when conservative pretty-printing is turned on; newly created layout is never inserted (see Section 4.3.2 about layout insertion).

BOX expressions are constructed by obtaining and instantiating pretty-print rules from pretty-print tables. Pretty-print rules can also be generated dynamically because all information required for rule generation is available in AsFix. Part of the functionality of the pretty-print generator `ppgen` is therefore reused in `pt2box` to dynamically generate pretty-print rules. This makes pretty-print rules dispensable because rules that are missing are generated on the fly.

Formatting abstract syntax trees   Generation of format trees from ASTs is implemented in `ast2box`. Given a set of pretty-print rules, `ast2box` produces the same format tree as `pt2box`, except for preserved layout strings. ASTs do not contain layout and consequently, conservative nor comment preserving pretty-printing is supported by `ast2box`.

Abstract syntax is generated from SDF definitions based on the constructor annotations of grammar productions. These constructor annotations define the node names of ASTs. See Chapter 2, "Grammars as Contracts", for a discussion about abstract syntax generation from concrete syntax definitions. Since constructor names also serve as keys in pretty-print tables, the name of a node in an AST can be used to obtain the corresponding pretty-print rule (see Figures 4.2 and 4.9).

In an AST, certain types of symbols are indistinguishable although they require different formatting. The symbol names in the keys of pretty-print rules serve to be able to always determine the types of symbols, such that the correct formatting can be applied.

For instance, SDF supports comma separated lists which have the same abstract syntax as ordinary lists. The separator symbols are not contained in the abstract syntax and have to be reproduced during pretty-printing. Consequently, comma separated lists require special treatment and should be distinguished form ordinary lists. This can be achieved with the information in the paths of corresponding pretty-print rules.

Keywords are not contained in ASTs and have to be reproduced during pretty-printing. ASTs also lack information to generate pretty-print rules containing these keywords dynamically. In contrast to `pt2box`, it is therefore essential for `ast2box` that pretty-print rules are defined for each constructor in the AST.

4.4.3   Format tree consumers

The last phase of pretty-printing consists of transforming a format tree to an output format. The architecture of GPP allows an arbitrary number of such format tree consumers. We implemented three of them which produce plain text, LaTeX, and HTML as output format, respectively. PDF can also be generated but indirectly from generated LaTeX code.

From BOX to text   The `box2text` component transforms a format tree to plain text. It fully supports comment preservation and conservative pretty-printing. The transformation consists of two phases. During the normalization phase all non-positional operators (except the C comment operator) are discarded, and positional operators are transformed to H and V boxes. A normalized BOX-term only contains non-conditional operators and absolute, rather then relative offsets as space options. Then, during the second phase, the normalized BOX-term is transformed to text. This amounts to producing non-

terminal symbols and layout strings. The latter are computed based on the absolute space options.

From BOX to LaTeX For the translation to LaTeX we defined LaTeX environments which provide the same formatting primitives of BOX in LaTeX [78]. The `box-2latex` consumer translates BOX-operators to corresponding environments, `latex` is then used to do the real formatting. An additional feature is the ability to improve the final output by defining translations from BOX strings to native LaTeX code. This feature can be used for example, to introduce mathematical symbols that were not available in the original source text (e.g., to introduce the symbol 'ϕ'' for the word `phi`). All code examples in this thesis (such as Figures 4.7–4.10) are generated with GPP using the box consumer `box2latex`.

From BOX to HTML Boxes are translated by `box2html` to a nested sequence of HTML tables. Representing BOX in HTML is difficult because HTML only contains primitives to structure text logically (as title, heading, paragraph etc.), not as composition of horizontal and vertical boxes. Only with HTML tables (where rows correspond to horizontal boxes and tables to vertical boxes) we can correctly represent BOX-operators in HTML. Figure 2.10 on page 32 and Figure 5.5 on page 85 contain examples of HTML pages as produced by `box2html`.

4.5 Applications

This section addresses several applications of the generic pretty-printer GPP in practice.

4.5.1 The Grammar Base

The Grammar Base (GB) is a collection of reusable Open Source language definitions in SDF. This collection includes grammars for XML, C, COBOL, JAVA, FORTRAN, SDL, and YACC.

In addition to language definitions, GB also contains pretty-print tables for each language. Together with generated parsers, GB thus offers a large collection of format tree producers and consumers for software reengineering systems for free.

The pretty-printer generator is used to generate initial pretty-print tables for new languages. By using this generator, pretty-print support can be guaranteed for each language without any effort. The generator is integrated in the build process of GB to automatically build a pretty-print table for a language unless a customized table exists.

We initiated the *Online Grammar Base* to make the grammars in GB accessible and browsable via Internet. GPP is used to produce the web pages by formatting all language definitions and representing them in HTML. The screen

dump in Figure 2.10 on page 32 gives an impression of the Online Grammar Base and the use of GPP as formatting engine.

4.5.2 Integration of GPP and GB in XT

GPP and GB are highly integrated in XT (see Chapter 3, "XT: a Bundle of Program Transformation Tools"), by general pretty-print tool pp. This tool combines all pretty-print components from GPP, with all pretty-print tables from GB. The result is a tool that can pretty-print any language contained in GB (currently more then 30 languages and 10 dialects) in any format supported by GPP (currently 3). The tool can format either PTs or ASTs.

With this tool, pretty-printing reduces to selecting an input language, an input format (plain text, PT, or AST), and an output format (plain text, HTML, or LaTeX).

4.5.3 A documentation generator for SDL

In the next chapter, we will discuss the development of a documentation generator for the Specification and Description Language (SDL) in corporation with Lucent Technologies. This documentation generator produces a web-site that provides different views on SDL programs and formed an application of GPP in industry.

We used ppgen to obtain an initial pretty-print table for SDL and added pretty-print rules to customize this generated formatting definition. We used pt2box to obtain a BOX format tree for SDL programs, containing labels and references for several language constructs. Depending on the view that was being constructed, a separate program connects the labels and references of relevant language constructs, for instance state transitions to corresponding state definitions. All HTML pages are produced by box2html.

We also developed utilities that produce BOX-terms for data types, such as lists. These tools made it very easy to transform such data types to HTML without the extra effort of developing an abstract syntax and pretty-print tables first. This illustrates that besides the format tree producers described thus far, also additional ones can easily be integrated with GPP.

The documentation generator also integrates other views of SDL programs, such as a graphical view on state transitions. These views can be accessed from the generated HTML representation of SDL programs and *vice versa*.

4.5.4 Pretty-printing COBOL

GPP is currently being used in an industrial experiment concerned with COBOL reengineering. The goal of this experiment is to bring the results of academic research into practice to build automated reengineering systems. The experiment combines *rewriting with layout* [32] and conservative, comment preserving pretty-printing to leave the layout of transformed programs in tact.

The transformations in this project are performed on full PTs and implemented in ASF [15, 54], an algebraic specification formalism based on rewriting. Transformation rules are interpreted by an engine that preserves layout during rewriting. Layout nodes that are created during rewriting are marked as new. The result is a PT that contains original layout nodes wherever possible, and nodes marked as new, where layout nodes had to be created. Consequently, only the parts of a COBOL program that were affected by the transformation are re-formatted by GPP, the rest of a program remains unchanged.

To use GPP in this experiment, we first generated a pretty-print table from the COBOL grammar using ppgen. Then we customized the generated pretty-print rules. Since pretty-printing occurs conservatively and the transformations make only local changes, few layout nodes have to be generated by the pretty-printer. Consequently, only a few pretty-print rules had to be customized to obtain a proper formatting of the transformed program parts.

Thus, although the COBOL grammar is large (about 350 nested productions) and the corresponding generated pretty-print table is huge (about 1200 pretty-print rules), only a few pretty-print rules had to be customized by hand (3 in the experiment that we performed) in order to use GPP successfully in this COBOL reengineering system.

4.6 Discussion

We only support conservative pretty-printing for PTs because AST do not contain layout. However, defining a tree format based on ASTs with layout, and to implement conservative pretty-printing for it, would not be difficult.[3]

We use pretty-print rule interpreters to transform PTs and ASTs into format trees. This has the advantage that the interpreters are language-independent and that pretty-printers can be customized at any time. Alternatively, language-specific format tree producers can be generated from pretty-print rules. This would increase performance at the cost of reduced flexibility.

In our approach, the rules that transform a PT or AST to a format tree are defined per language production. This transformation can also be based on pattern matching in which case the tree structure is taken into account to determine a proper formatting of program fragments [33, 106]. This gives more expressiveness because it is context sensitive. However, reuse of format definitions for formatting PTs and ASTs would be complicated due to the usually different tree structures.

---

[3]See for instance [148], which discusses layout preserving reengineerings defined on abstract syntax trees with layout.

4.7 Concluding remarks

Related work   We refer to [79] for a general overview of existing work in pretty-printing.

An early approach to comment preserving pretty-printing was discussed in [26]. In this approach, the original program text is needed during pretty-printing in order to retrieve comments and to determine the location where to insert comments in BOX-like formatting expressions.

Van De Vanter emphasizes the importance of comments and white space for the comprehensibility and maintainability of source code [136]. Like we, he advocates the need to preserve this crucial aspect. He describes the difficulties involved in preserving this documentary structure of source code, which is largely orthogonal to the formal linguistic structure, but he does not provide practical solutions.

In [125] another conservative pretty-print approach is described which only adjusts code fragments that do not satisfy a set of language-specific format constraints. This approach is useful for improving the formating of source code by correcting format errors. However, it cannot handle completely unformatted (generated) text and does not support comment preservation. Furthermore, it adapts existing layout (in case format errors are detected), which makes it not useful in general for software reengineering.

Pretty-printing as tree transformation is also discussed by [67, 106]. They do not address typical requirements for software reengineering such as customization and reuse of transformation rules, and layout preservation. Also transforming different types of trees and producing different output formats is not addressed.

Components and Reuse   Figure 4.12 displays the gpp package, its constituent components, and the components it reuses (see Section 3.6 on page 42 for information about component diagrams). The gpp package implements 8 components and reuses 6 components from 6 different packages. The reused components are part of XT and are described in more detail in Chapter 3, "XT: a Bundle of Program Transformation Tools".

Table 4.2 depicts component sizes and reuse levels of the gpp package. The table shows that the implementation consists of approximately 12,100 lines of code, of which more than 8,400 lines are reused. This yields a reuse level between 69% and 85%. Section 3.6 justifies these numbers and describes how they are obtained by analyzing component implementations.

Contributions   In this chapter, we discussed pretty-printing in the context of software reengineering and described techniques that make pretty-printing generally applicable in this area. These techniques help to decrease development and maintenance effort of reengineering systems, tailored towards particular customer needs, and supporting multiple language dialects. The tech-
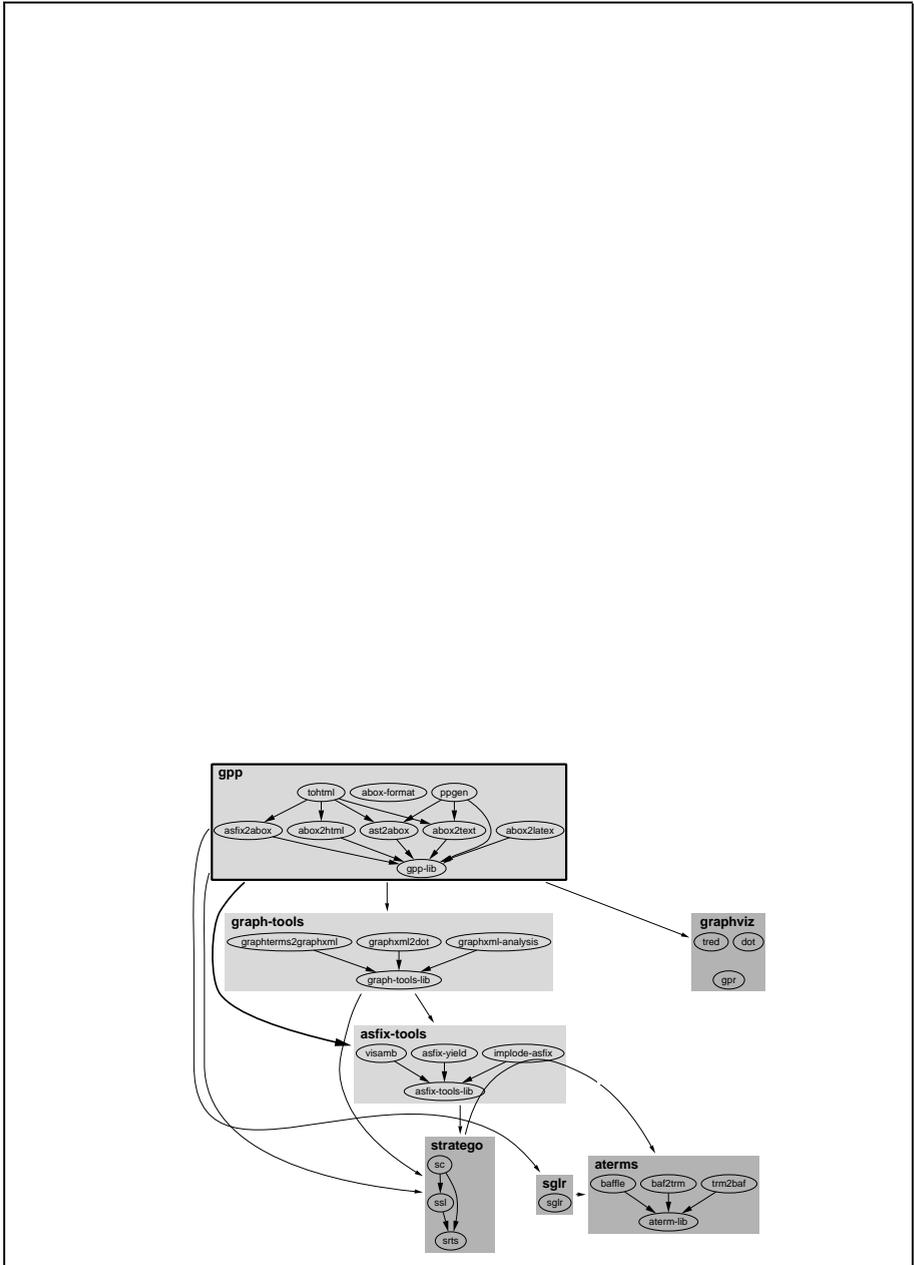
Figure 4.12 Components used by the GPP package. This picture will be filled in the remaining chapters with additional software packages. This leads to a complete picture of component usage, which is depicted in Figure 8.2 on page 145.

| | Non-transitive reuse | | | Transitive reuse | |
Component	LOC	RSI	Reuse%	LOC	Reuse%
abox2html	1,436	783	54%	1,436	54%
abox2latex	1,782	895	50%	1,782	50%
abox2text	1,226	805	65%	1,226	65%
abox-format	755	676	89%	755	89%
asfix2abox	3,617	2,883	79%	5,325	86%
ast2abox	1,714	1,382	80%	3,422	90%
ppgen	1,415	984	69%	1,415	69%
tohtml	233	0	0%	10,988	97%
Totals:	**12,178**	**8,408**	**69%**	**26,349**	**85%**

Table 4.2 Reuse table for the gpp package. The table shows that for this package, a total of 3,770 *new* lines of code had to be written.

niques addressed in this chapter include i) *customizable pretty-printing* to adapt a pretty-printer to customer-specific format conventions, ii) *conservative pretty-printing* to preserve layout and comments during pretty-printing, iii) *modular pretty-print tables* to make formattings reusable, and iv) pretty-printing of PTs *and* ASTs using a single format definition. Furthermore, we implemented these techniques in the generic pretty-printer GPP which transforms PTs and ASTs to plain text, HTML, and LaTeX, using a single format definition. Finally, we demonstrated that these techniques are feasible by using GPP for a number of non-trivial applications, including COBOL reengineering and SDL documentation generation.

Development with Reuse

Cost-Effective Maintenance Tools for Proprietary Languages

This chapter addresses Language-Centered Software Engineering (LCSE) in practice using the techniques and language tool components presented in Chapters 2–4.

This chapter describes a case study, carried out in cooperation with Lucent Technologies, concerned with decreasing the maintenance costs of proprietary language tooling. The case study was concerned with Lucent's proprietary SDL *dialect and involved reengineering their dialect and developing a documentation generator for it. We show that by using LCSE and the components from the* XT *bundle, the development process of languages and tools can be simplified and a decrease in maintenance costs can be achieved. Further, we show that the development time of the documentation generator was relatively short because a significant amount of code was reused as-is or generated by the program generators that are bundled with* XT*. The work presented in this chapter was published earlier as [82].*

5.1   Introduction

Many companies in industry use proprietary programming languages or language dialects for the development of their software products. Consequently, such companies are faced with a maintenance effort for their software products *and* for the language and corresponding tooling. Maintenance costs of the language and tooling are usually high because a language change has great impact on the corresponding tooling [55].

To overcome such maintenance problems of language tools, a company has four options:

1. It might decide to stop the development of the language and tooling. This is a short term solution however, because knowledge of the tooling will gradually disappear which eventually may lead instead to an increase of maintenance costs.

2. Migrate its software products and implement them in a similar, international, standardized programming language (if available). After migration, commercially available language tooling can be used and the company can benefit from main stream language development. Of course, this approach can be very difficult, and may also have great impact on the development process of the software products and its developers, depending on the amount of discrepancies between the standard and the proprietary language.

3. Outsource the development and maintenance of its language tooling. This is a transfer of the problem to a third-party, the maintenance problem still remains. Furthermore, it usually is extremely expensive and it makes the company dependent on a third party.

4. Decrease maintenance costs by simplifying the development process of language tooling. This can be achieved, either by increasing the amount of generated, language-specific code, or by increasing the number of reused (language-independent) components.

In this chapter we explore the last option. We describe a case study carried out in co-operation with Lucent Technologies in which we used Language-Centered Software Engineering (LCSE) as discussed in Chapter 2, "Grammars as Contracts", for the development of maintainable, inexpensive language tools for incremental documentation generation.

This case study was concerned with Lucent's proprietary dialect of the Specification and Description Language (SDL), a language for the specification of the behavior of telecommunication systems. Lucent based its dialect on an early draft of the international standard, which only slightly differed from the final standard which appeared in 1988 [72]. To limit maintenance and development costs, Lucent Technologies decided long ago to 'freeze' their proprietary SDL dialect. Since adaptations to their SDL tool set were no longer required, knowledge of these tools disappeared gradually. This eventually led to increased development costs of new language tools and unavailability of existing tools due to the dependency on now obsolete hardware. Clearly, Lucent's early approach of reducing maintenance costs by freezing language and tool development has turned out to be counter-productive on the long term.

To demonstrate that LCSE simplifies language tool construction and that it can decrease maintenance costs, we started the development of a new tool

environment for code browsing and visualization. This environment helps maintaining Lucent's SDL code because it improves accessibility of SDL code by providing access to the source code at different levels of abstraction and from different points of view. Since the environment can easily be extended, maintenance of SDL code can further be improved by connecting additional documentation and visualization components. Without preliminary knowledge about SDL, we developed a prototype tool environment in only a few man months time. After building some basic SDL components together, Lucent constructed its own language tooling using the techniques described in this chapter.

The chapter is organized as follows. We address LCSE in Section 5.2. That section motivates, amongst other things, the use of SDF for syntax definition. Section 5.3 discusses how an SDF grammar for SDL can be reengineered from an operational YACC grammar. This SDF grammar is used in Section 5.4 where we develop an SDL documentation generator for code browsing and inspection. Sections 5.5 and 5.6 contain a discussion of related work, a summary of contributions, and directions for future work.

5.2   Language-centered software engineering[1]

Grammars form an essential part of language tools, not only because they are used to derive parsers and pretty-printers from, but also because they greatly influence the shape of corresponding parse trees. Consequently, all parts of a software system that operate on parse trees depend on the grammar.

Different language tools perform several common sub-tasks, such as parsing, pretty-printing, and tree traversal. To optimize the software development process by minimizing code duplication, it is necessary that the code performing these tasks is shared between different language tools. Moreover, being free to choose a programming language that best suits the needs of an application is necessary as well. Sharing functionality between applications based on source code reuse only is therefore not sufficient. In addition to source code reuse, component-based software construction is required to share common tasks between applications written in different programming languages.

LCSE, which was presented in Chapter 2, emphasizes the central role of grammars and the need for components in the software engineering process. It supports generation of stand-alone components and libraries from grammars as well as easy integration of off-the-shelf reusable components. LCSE is based on the following key concepts.

Contracts   Building applications by connecting reusable, generated, as well as application-specific components requires agreement on the type and structure of data exchanged between these components. Further, a uniform exchange

---

[1]Language-Centered Software Engineering (LCSE) was discussed before in Chapter 2, "Grammars as Contracts", and in Chapter 3, "XT: a Bundle of Program Transformation Tools". This section contains a brief overview of the techniques and tools presented there.
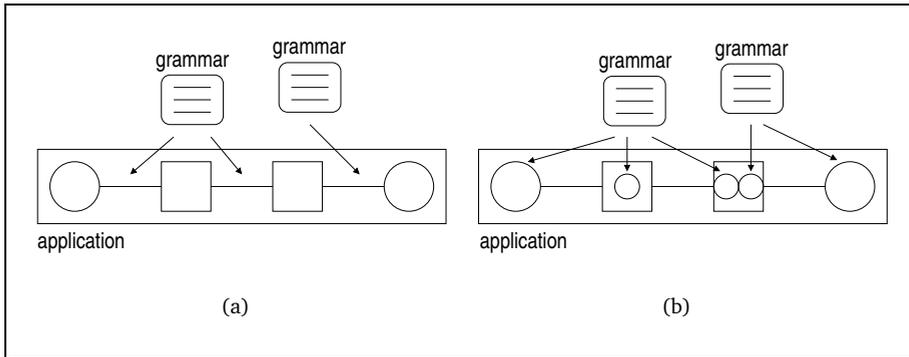
Figure 5.1 Grammars serve as contracts between different components (a) and for generation of language-specific code (b).

format is needed in order to connect such components easily. Since language tools typically transfer parse trees or abstract syntax trees between different components, the language itself describes the structure of the data that is transferred. In Chapter 2 we described an architecture of component-based software development where grammars serve as *contracts* between components (see Figure 5.1(a)).

Library generation Since the structure of trees processed by language tools depends on the grammar, the code to traverse such trees also depends on the same grammar. Obviously, writing such code by hand is time consuming, error prone, and yields a maintenance problem because this code needs to be adapted over and over again whenever the grammar changes. Moreover, this programming effort has to be repeated for each programming language in use. Generation of libraries containing tree access and traversal functions from the grammar for each programming language in use (the small circles in Figure 5.1(b)) therefore helps cost effective language tool development.

Component generation In addition to generating programming language-specific libraries, a grammar can also be used to generate language-specific, stand-alone components, which can be reused as-is to construct new applications (the large circles in Figure 5.1(b)). Such components include parsers and pretty-printers, as well as tools to convert parse trees into abstract syntax trees and *vice versa*.

Language technology LCSE requires state-of-the-art language technology for language definition and parsing. Reusability and maintainability of grammars are essential to fully benefit from LCSE. To meet these requirements, language technology is needed that:

- Accepts the full class of context-free grammars. This allows for clear encodings of languages, which do not have to be manipulated to fit in a restricted class of grammars.

- Offers a purely declarative syntax definition formalism. This prevents pollution of grammars with (application-specific) semantic actions that would hamper reuse and maintainability of grammars.

- Supports modular syntax definitions. Modularization allows language dialects to be defined as grammar extensions in separate modules. This prevents duplication of grammar definitions.

These requirements are fulfilled by the syntax definition formalism SDF [68, 137] together with generalized LR parsing [122, 137]. We used this technology in the case study presented here for SDL syntax definition and parsing.

LCSE helps to decrease the development time of language tools because it minimizes the amount of language-specific code that has to be written by hand. Once a grammar exists, language-specific, stand-alone components and libraries are generated, which, together with off-the-shelf reusable components, help to build new language tools easily. LCSE also improves software maintenance because due to code generation and component reuse, only a relative small part of an application requires manual adaptation after a language change.

LCSE is supported by the tool bundle XT, which we presented in Chapter 3. XT, which stands for 'Program Transformation Tools', bundles various related transformation tool packages into a single distribution. These packages include a generalized LR parser and parser generator [137], The ATERMS uniform exchange format [28], the transformational programming language Stratego [138], and the generic pretty-printer GPP (see Chapter 4, "Pretty-Printing for Software Reengineering"). Furthermore, XT contains an extensive collection of grammar related tools and the Grammar Base, a collection of reusable grammars. XT is distributed as open source and requires minimal installation effort.

5.3 SDL grammar reengineering

In Section 5.2 we discussed the requirements that LCSE puts on language technology and we motivated the use of SDF and generalized LR parse techniques. To benefit from these in order to build maintainable tools for SDL, we derived an SDF grammar from the EBNF definition and operational YACC grammar that were available for Lucent's SDL dialect. This section describes the systematic process that we followed to obtain a cleaned up grammar of this SDL dialect in SDF.
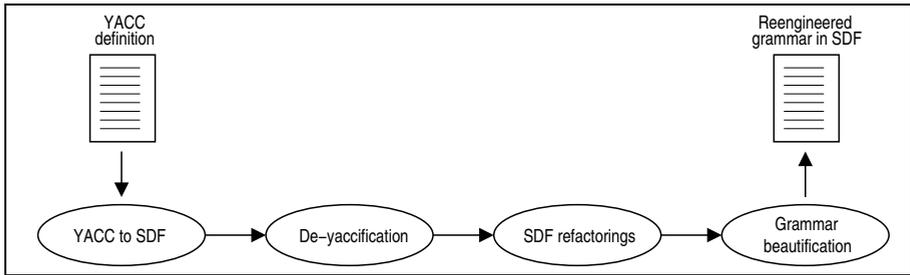
Figure 5.2 The four phases of the grammar reengineering process in which an SDF definition is obtained from a (legacy) YACC definition.

5.3.1 From YACC to SDF

Although SDF in combination with LR parsing offers advanced language technology which decreases maintenance costs of grammars and promotes their reuse, far more grammars have already been developed in YACC, many of which are operational and extensively tested. Since the development of a grammar from scratch is difficult and expensive, we propose systematic grammar reengineering, in which an SDF grammar is (semi-automatically) obtained from an operational (legacy) YACC grammar, yielding a grammar that is as correct as the originating one.

   We divided this grammar reengineering process in four phases in order to make a clear distinction between different types of transformations (see Figure 5.2). During the first transformation phase, an SDF grammar is obtained from the YACC grammar. The remaining phases define source to source transformations on SDF in which a more natural encoding of the language is obtained. Since only the first two phases of the reengineering process are YACC-specific and only the first phase actually depends on the YACC syntax, this reengineering approach can also be defined for other syntax definition formalisms (like BNF and ANTLR [117]). Only a front-end which transforms a syntax definition to SDF, and a transformer which removes system-specific constructs need to be defined.

YACC to SDF phase The YACC to SDF phase is completely automated and basically consists of the following transformation:

$$
\begin{array}{rl}
S & : \quad S_i \ldots S_{i+k} \\
  & | \quad S_j \ldots S_{j+n} \\
  & \vdots \\
  & ;
\end{array}
\qquad \Longrightarrow \qquad
\begin{array}{rcl}
S_i \ldots S_{i+k} & \to & S \\
S_j \ldots S_{j+n} & \to & S \\
\vdots & &
\end{array}
$$

Observe that each alternative in the YACC definition yields a separate production in SDF, and that the productions are reversed with respect to formalisms

like BNF. Furthermore, since SDF only allows syntax definition, all semantic related issues from the original YACC definition are removed during this phase. This includes removal of YACC's error handling mechanism using the reserved token "error".

Since we are interested in reengineering grammars only, we ignore semantic actions here. This holds for semantic actions that control the shape of parse trees as well as for semantic actions related to error recovery and error reporting. We use a generic parse tree format and consider modifying the shape of parse trees for particular needs as a separate phase *after* parsing. For error reporting and recovery we depend on generic mechanisms provided by our parser.

De-yaccification phase   An SDF grammar specification obtained in the previous phase is isomorphic to the grammar defined in YACC and therefore still contains YACC-specific idioms. These include lists expressed as left-recursive productions and productions introducing non-terminal symbols which only serve disambiguation (by encoding precedence and associativity in grammar productions).

In order to obtain a more natural specification of the language in SDF, we remove these YACC idioms, a process called de-yaccification [127]. We consider the following transformations:

- Introduction of ambiguous productions by flattening the productions that only serve disambiguation. This transformation also includes the addition of priority rules for disambiguation.

- Replacement of left-recursive list encodings by explicit list constructs.

- Joining lexical and context-free syntax by *unfolding* terminal symbols in context-free productions.

These transformations are performed automatically and have been implemented in the algebraic specification formalism ASF + SDF [15, 68, 54], a formalism supporting conditional rewrite rules based on concrete syntax. Its recent extension with traversal functions simplified the construction of transformation systems significantly [29]. The system provides default bottom-up traversals and the programmer provides rewrite rules only for those constructs that need transformation. Since the individual de-yaccification transformations only affect a small part of the SDF grammar (which itself is quite large), only a few rewrite rules are needed to implement the transformations. Each transformation is defined as a separate ASF + SDF specification which can be executed in sequence to remove YACC-specific idioms step by step.

SDF refactoring phase   After the de-yaccification process, the grammar looks already more natural. The grammar still does not benefit from the powerful features of SDF however. The next step in the reengineering process is to

```
        decision      : decisionStart decisionBody ENDDECISION
                      | decisionStart error
                      ;
        decisionStart : DECISION decisionValue endStmt
                      | DECISION error
                      ;
        decisionBody  : caseList
                      | caseList elseAnswer caseTransition
                      ;
        caseList      : case
                      | caseList case
                      ;
```

Figure 5.3 YACC fragment of SDL Decision construct.

refactor the grammar to introduce such special SDF constructs which shorten the syntax definition and improve its readability. We consider the following transformations:

- Introduction of optionals.

- Introduction of SDF constructs for separated lists.

- Grammar decomposition by modularization.

We developed ASF + SDF specifications to automate the first two transformations. We do not have tool support for automatic modularization of grammars yet, but heuristics for grammar decomposition are described in [122, 128].

Beautification phase The previous transformations are general language-independent grammar transformations most of which can be automated. The transformations operate globally on the grammar and transform productions that match general patterns.

During the last phase of the reengineering process, the grammar can be fine tuned by performing transformations operating on specific grammar productions. What transformations to perform largely depends on personal taste. An operator suite designed for expressing this kind of grammar transformations and applying them automatically is described in [94, 90].

5.3.2 SDL **grammar reengineering**

Lucent's proprietary SDL dialect is closely related to the SDL standard known as SDL 88 [72]. It was derived from an early (yet incomplete) draft of this standard. The absence of *block* constructs in the SDL dialect was the major

difference with SDL 88. After SDL 88 many new language constructs have been defined in additional SDL standards of which SDL 2000 is the most recent one. Lucent's dialect did not benefit from these language progressions.

Both the syntax and semantics of Lucent's proprietary SDL dialect have been clearly defined in internal technical reports. Like standard SDL, the dialect exists in two forms: a graphical form (SDL-GR) and a textual form (SDL-PR). Since a mapping exists between both, we only considered the textual representation for the reengineering project that we carried out. The lexical and context-free syntax of this language was defined in a single EBNF definition. In addition to the technical documentation we also had an operational YACC definition available which has been in use for years in the original SDL toolset. We considered this definition as ultimate starting point for reengineering the SDL grammar. Unfortunately, the lexical analyzer that was also available turned out to be useless for this reengineering process because the lexical syntax was directly coded in C procedures and consequently, hard, if not impossible, to reengineer.

The reengineering process therefore consisted of a (semi-) automatic derivation of the context-free syntax from the YACC definition, and a manual definition of the lexical syntax. Thanks to the clear EBNF definition the mapping from the EBNF lexical syntax definition to SDF was straight forward (it was only a matter of minutes) and is not further addressed here.

After the transformation to SDF was complete, we added constructor names to the context-free productions of the SDL grammar which are used to derive an abstract syntax definition (see Section 5.4). This annotated SDL grammar, allows tooling to operate on full parse trees (for example comment preserving pretty-printers), and on abstract syntax trees.

While testing the reengineered grammar, we discovered that in existing SDL code slightly different lexical constructs were used as were defined in Lucent's EBNF definition. Since SDF definitions are modular, we were able to develop an extension to the SDL lexical syntax in a separate module to accept these constructs *without* affecting the lexical syntax definition that corresponds to the EBNF definition.

The reengineering process transformed the YACC fragment of Figure 5.3 to the SDF fragment of Figure 5.4. The semantic actions that were attached to most of the syntax productions in the original YACC definition are not shown. Observe that the reengineered fragment contains fewer productions, that keywords are contained in the productions, and that SDF constructs such as optionals and lists are introduced.

The complete reengineered SDL grammar is defined in 23 modules each defining particular SDL constructs. The number of non-terminals has been reduced from 254 to 104, the number of productions from 490 to 190.

```
context-free syntax
  "DECISION" DecisionValue ";" DecisionBody "ENDDECISION"
      → Decision {cons("Decision")}

  Case+ ( ElseAnswer CaseTransition )?
      → DecisionBody {cons("DecisionBody")}
```

Figure 5.4 Reengineered SDL Decision construct of Figure 5.3 in SDF.

5.4 An SDL documentation generator

This section addresses the development of an extensible documentation generator for SDL in textual form (SDL-PR). We did not consider SDL in graphical form (SDL-GR). The generator produces HTML documentation from SDL code. It uses extractors to collect specific information from SDL code which is used to provide different ways for code browsing. The documentation consists of a state name list and a pretty-printed SDL program. When clicking on a state name, the corresponding state definition is presented. A screen dump of generated documentation is depicted in Figure 5.5.

The documentation generator is based on the SDL grammar as developed in the previous section. From this grammar, we generate a parser which produces parse trees in the SDF parse tree format, called AsFix [137]. AsFix trees contain all information about parsed terms, including layout and comments. This enables the exact reconstruction of the original input term and the ability to process parse trees while preserving comments and layout (see Chapter 2, "Grammars as Contracts"). From the grammar also an abstract syntax definition can be derived, as well as pretty-print tables and Stratego signatures (see below).

Abstract syntax From the concrete syntax definition, an abstract syntax definition can be derived based on the prefix constructor names defined as annotations of grammar productions (the cons attributes in Figure 5.4). These constructor names define the names of abstract syntax productions and can be added to the grammar manually or be synthesized automatically using the sdfcons tool (see Chapter 2, "Grammars as Contracts"). Abstract syntax trees can be derived from AsFix parse trees because the constructor name information is contained in AsFix trees. Language tooling can be developed to operate on parse trees or on abstract syntax trees depending on particular needs. The components of the documentation generator that we implemented operate on abstract syntax trees because they only require a subset of the information that is contained in the parse trees. The comment preserving pretty-printer that we reuse for pretty-printing SDL code on the other hand operates on parse trees.
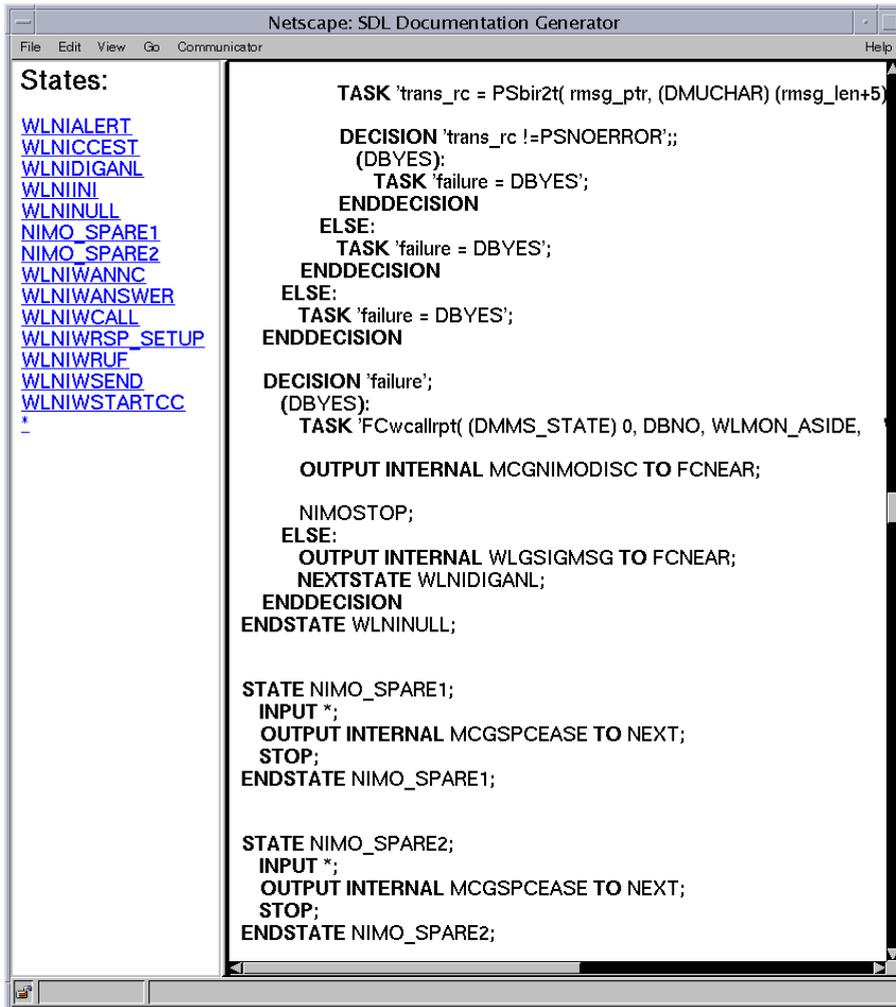
Figure 5.5   Generated SDL documentation.

Pretty-print tables   Parse trees and abstract syntax trees can be transformed into human-readable form by the generic pretty-printer GPP. This pretty-printer can produce several formats including plain text, LaTeX, and HTML. Precise, language-specific formattings are expressed in pretty-print tables which define mappings from language constructs (denoted by their constructor names) to BOX. GPP and BOX were discussed in Chapter 4, "Pretty-Printing for Software Reengineering".

BOX is a language-independent markup language designed to define the intended formatting of text. Pretty-print tables can be generated from a grammar
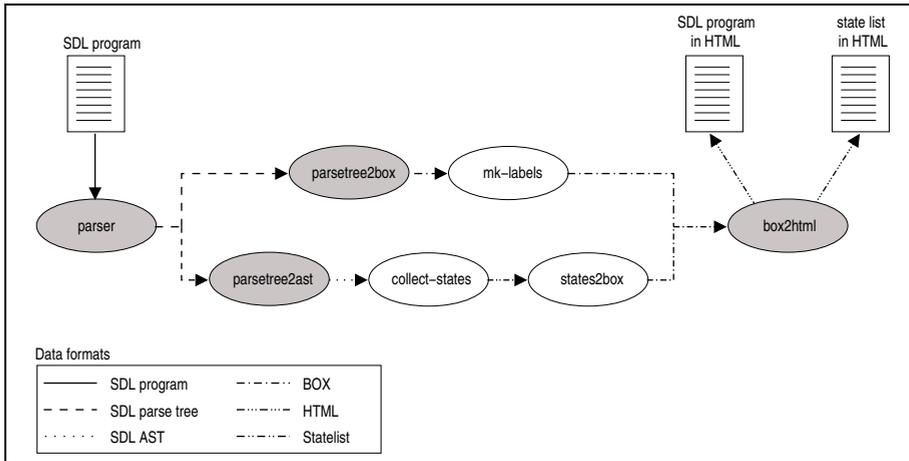
Figure 5.6 SDL documentation generation process.

and customized to define the desired formatting. For example, a formatting for the Decision construct of Figure 5.4 can be specified using the following pretty-print rule:

Decision **-- V [H [**"DECISION" _1 ";"**]** _2 "ENDDECISION"**]**

This rule specifies that the keyword DECISION, the non-terminal DecisionValue and the semi-colon should be formatted horizontally. This horizontal box, together with the box representing the pretty-printed non-terminal DecisionBody, and the keyword ENDDECISION should be formatted vertically.

BOX supports labels in pretty-print rules which are translated to HTML anchors by the `box2html` formatter. With this labeling mechanism, we can add links to SDL language constructs in the generated HTML documentation. For the SDL documentation generator we are interested in state definitions such that we can jump in SDL code to the start of state definitions. Therefore, we use the LBL construct of BOX to define labels in the pretty-print rule of the State construct:

State **-- V [H [**"STATE" **LBL [** "STATE" _1 **]** ";"**]** _2 **H [**"ENDSTATE" _3 ";"**]]**

Afterwards, an SDL-specific tool (`mk-labels`) replaces the label names as generated by the box generator (STATE in the example above) by unique names. Since the pretty-printer is language-independent it cannot be used here for this language-specific synthesis of label names.

After the generated pretty-print table for SDL has been customized to define appropriate formatting and labels for state definitions, generic tooling can be used to translate a parse tree to BOX and subsequently to one of the output

formats. For the documentation generator we only use HTML, but the generation of documentation with richer formatting is supported via the box2latex back-end.

Stratego   We used the programming language Stratego for the implementation of the SDL-specific components. Stratego [138] is a program transformation language based on term rewriting with strategies. It has an extensive library of strategies for term traversals and transformations. Stratego also supports the common exchange format ATERMS, which enables processing of parse trees and abstract syntax trees as produced by the SDL parser. In order to enable special term traversals and transformations, language-specific signatures are required by Stratego. These signatures which describe the shape of abstract syntax trees are generated from an SDF definition by the tool sdf2sig.

The SDL-specific mk-labels tool, which inserts unique label names in a BOX term to denote the start of state definitions in SDL programs, is implemented in Stratego as follows:

```
module mk-labels.r
strategies
  mk-labels = topdown(try(mk-label))

  state-name2str = getfirst(mk-string)
rules
  mk-string : S(str) → str

  mk-label : LBL("\"STATE\"", abox) → LBL(name, abox)
            where
               <concat-strings>["state:", <state-name2str> abox] ⇒ name
```

This program performs a top-down traversal of a BOX term, and tries to apply the rule mk-label to each node. This rule replaces the first argument of label nodes of the form LBL("\"STATE\"", abox) by the state name, preceded by the string "state:". The state name is retrieved from the second argument of the LBL term using the strategy state-name2str. This strategy retrieves the first of a list of names that is given to a state definition, and makes a string from it. The program thus transforms a term

```
LBL("\"STATE\"", [S("my_state")])
```

into the term

```
LBL("state:my_state",[S("my_state")])
```

Once appropriate label names have been inserted by the tool mk-labels, the BOX term can be translated to HTML using the box2html back-end. The resulting HTML pages contain anchors at the start of state definitions.
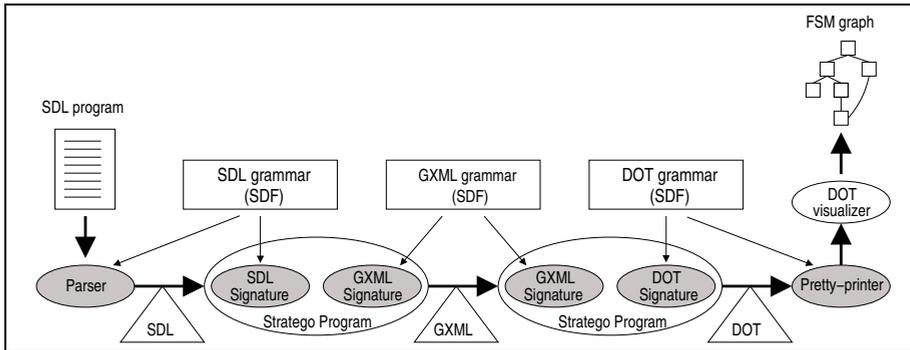
Figure 5.7 Architecture of the FSM graph generator.

Extraction The documentation generator generates a web-site which displays an SDL program to the user and allows him to browse this code in different ways. One way to browse the code is by offering a list of state names which, when clicked on, jump to the start of the corresponding state definition. To implement this, a code extractor needs to be implemented to collect a list of states from an SDL program. This is easily implemented in Stratego by traversing over the abstract syntax tree and collecting all nodes that correspond to state definitions:

```
module collect-states
strategies
  collect-states = collect(get-state-name)
rules
  get-state-name : State(StateList(names), _, _) → <Hd> names

  get-state-name : State(StateList(), _, _) → "⋆"
```

This program matches state nodes and distinguishes ordinary state definitions and default states ('⋆' states). For ordinary state definitions, the first of the list of state names is returned (with <Hd>names). For default states, '⋆' is returned as state name.

Collection of states is a simple extraction but when combined with other (more advanced) extractors, a rich SDL toolset can be constructed. A more complex example is the state transition extractor which is discussed later.

Connection to the documentation generator To integrate the state collector in the documentation generator, its output should be represented in HTML as a list of hyper-links each pointing to the start of the corresponding state definition in a pretty-printed SDL file. These hyper-links should use the same label names as generated by the mk-labels tool described above. We consider transforming the output of the state collector to HTML as a separate step to be

performed by a separate component. Therefore, the state collector does not produce HTML directly and, as a result, can also be used in different settings.

To obtain an HTML representation of the list of states, we can construct a grammar for the output format and a pretty-print table to define the mapping to HTML. Because this approach has some overhead (it requires an additional grammar and extra parsing of the list of states), we followed a different approach and developed a small component that transforms the output directly to a BOX term (more precisely it produces an abstract syntax tree of a BOX term). This term can be passed to `box2html` to obtain the desired HTML representation.

Integration of components All ingredients of an initial documentation generator have now been developed. We can format SDL programs in HTML by parsing a program and transforming the parse tree to BOX using the SDL pretty-print table. Then we can insert unique label names in the BOX term and transform it to HTML. The HTML list of state names (which link to state definitions) can be obtained by first parsing an SDL program and transforming the parse tree to an abstract syntax tree. Then we can collect all state names and translate the resulting list of states to BOX and finally to HTML. Figure 5.6 contains an overview of all components involved in this documentation generation process and shows how both the HTML state list and the pretty-printed SDL code are produced. Grey ellipses denote generated or reused components.

A FSM graph generator The documentation generator is extendible and extra tooling can be developed to provide additional documentation and information of SDL programs. In addition to the state collector, we developed a finite state machine (FSM) graph generator. This generator produces, given an SDL model, the graph representation of the underlying FSM. In Figure 5.7 a detailed overview of the graph generator is depicted. Grey ellipses denote the components that are generated.

In addition to the SDL grammar, two more grammars are used for this tool. DOT [63] is a low level graph representation, which we used because off-the-shelf graph visualization tools for this representation were available for reuse. GRAPHXML [70] is a high-level graph representation language in XML in which a graph can be represented in terms of its mathematical description (i.e. in terms of edges and transitions). The grammars for DOT and GRAPHXML are available in the Grammar Base and reused here as off-the-shelf *language components*.

The only thing that needs to be implemented for the FSM visualizer is part of a single Stratego program (the left-most Stratego program in Figure 5.7), which is responsible for the extraction of the FSM information from SDL code and the generation of a graph representation in GRAPHXML. The tool that transforms GRAPHXML to DOT and the DOT visualizer were reused as-is. Both Stratego components share the generated grammar signatures. In Figure 5.8

a generated FSM graph is depicted that was extracted from a real-world SDL program of approximately 30,000 lines of code. This figure shows transitions from states (denoted as ellipses) and from procedures (denoted as diamonds). The corresponding procedure call graph was extracted in a similar fashion and is depicted in Figure 5.9.

The FSM state generator is another example of an extractor and can easily be integrated in the SDL documentation generator. This is achieved by automatically converting the graph into a clickable image such that clicking on a node in the graph jumps to the corresponding state definition.

Maintainability   Thanks to component reuse and code generation, the documentation generator could be implemented with little programming effort. All components together required 165 lines of Stratego code.[2] Maintenance costs of these components is low because due to the generic term traversals of Stratego, language dependence of the components is limited, reducing the amount of code that needs to be adapted after a language change. For instance, the `collect-states` tool only depends on two SDL language constructs related to state definitions. Only when these constructs are changed in the language, the tool needs modification. Together with the modularization mechanism of SDF, this also greatly simplifies the simultaneously development of components for multiple SDL dialects. Additionally, component reuse and code generation also makes extending the generator relatively easy.

5.5   Related work

In [96], a semi-automated grammar recovery project is described where a complete grammar for VS COBOL II is constructed from an online manual. Grammar recovery from BNF definitions is discussed in [128]. In contrast to our approach based on an operational YACC definition, these approaches require grammar correction because they are based on non-operational language descriptions which often contain errors. A reengineering approach similar to ours, not requiring grammar correction is described in [127]. They also derive an SDF grammar from an operational YACC definition but their approach yields grammars which are not optimal for software development. Heuristics for de-yaccification are described in [150]. The authors focus on abstract syntax derivation from concrete syntax which benefits from a clear natural encoding of a language. Formalization of grammar transformations is addressed in [94]. They describe an operator suite for grammar adaptation which is derived from a few fundamental grammar transformations and supplemental notions like focus, constraint, and yielder.

In addition to the SDL grammar and bottom-up, generalized LR parser that we described here, the development of a *top-down* parser for SDL 2000 using

---

[2]These are real-written lines of Stratego code (i.e., they are not normalized or pretty-printed as was done to obtain the numbers in Table 5.1 on page 93).

Figure 5.8 A generated FSM graph depicting transitions from states (denoted by ellipses) and from procedures (denoted by diamonds). The corresponding procedure call graph is depicted in Figure 5.9.

Figure 5.9 Call graph corresponding to the FSM graph of Figure 5.8.

ANTLR [117] as parser generator is described in [126]. Another approach using
recursive descent parsing with backtracking is described in [49].

Component	Non-transitive reuse			Transitive reuse	
	LOC	RSI	Reuse%	LOC	Reuse%
`cluster`	898	775	86%	898	86%
`collect-states`	806	773	95%	806	95%
`collect-transitions`	897	788	87%	897	87%
`mk-labels`	730	673	92%	730	92%
`sdl-doc`	45	0	0%	17,821	99%
`states2abox`	608	571	93%	608	93%
`transitions2gxml`	1,059	907	85%	1,059	85%
Totals:	**5,043**	**4,487**	**88%**	**22,819**	**97%**

Table 5.1 Reuse table for the sdl-tools package. The table shows that for this package, a total of 556 *new* lines of code had to be written.

In addition to XT, many environments and tools exist for program transformation. The online survey of program transformation [143] strives to give a comprehensive overview of program transformation and transformation systems.

Hypertext for software documentation is discussed in several papers [38, 120, 25, 121]. The SDL documentation generator, presented in this chapter, was inspired by DOCGEN [57], a generator for interactive, hyperlinked documentation about legacy systems. They use Island Grammars (i.e. partial syntax definitions) for code extraction instead of full grammars as we do. A less precise extraction approach based on lexical analysis only is discussed in [107].

5.6   Concluding remarks

Contributions   This chapter addressed grammar reengineering and the construction of maintenance tools for proprietary languages and dialects. The chapter demonstrates that Language-Centered Software Engineering (LCSE) decreases development time of such language tools: once an SDF grammar for SDL was developed, implementing the tools described in this chapter required only limited effort. This is because language-dependent components and libraries are generated and because existing (third-party) components can be used and integrated easily. Semi-automatic grammar reengineering brings LCSE into reach because it significantly reduces the effort to move to essential state-of-the-art language technology. We demonstrated that modular syntax definitions, the generation of language-specific code, and language independence of Stratego programs help maintaining multiple language dialects. At the time of this writing, the techniques presented in this chapter are being

used within Lucent Technologies to further develop the SDL documentation generator and related tools.

Components and reuse   The SDL documentation generator is contained in the sdl-tools package. Figure 5.10 displays this package, its constituent components, and the components it reuses (see Section 3.6 on page 42 for information about component diagrams). The sdl-tools package implements 7 components and reuses 12 components from 13 different packages. Observe that this figure extends Figure 3.1 on page 47 with additional SDL-related components. This demonstrates reusability and compositionality of components developed following the LCSE model.

Table 5.1 depicts component sizes and reuse levels of the sdl-tools package. The table shows that the implementation consists of approximately 5,000 lines of code, of more than 4,400 lines are reused. This yields a reuse level between 88% and 97%. Section 3.6 justifies these numbers and describes how they are obtained by analyzing component implementations.
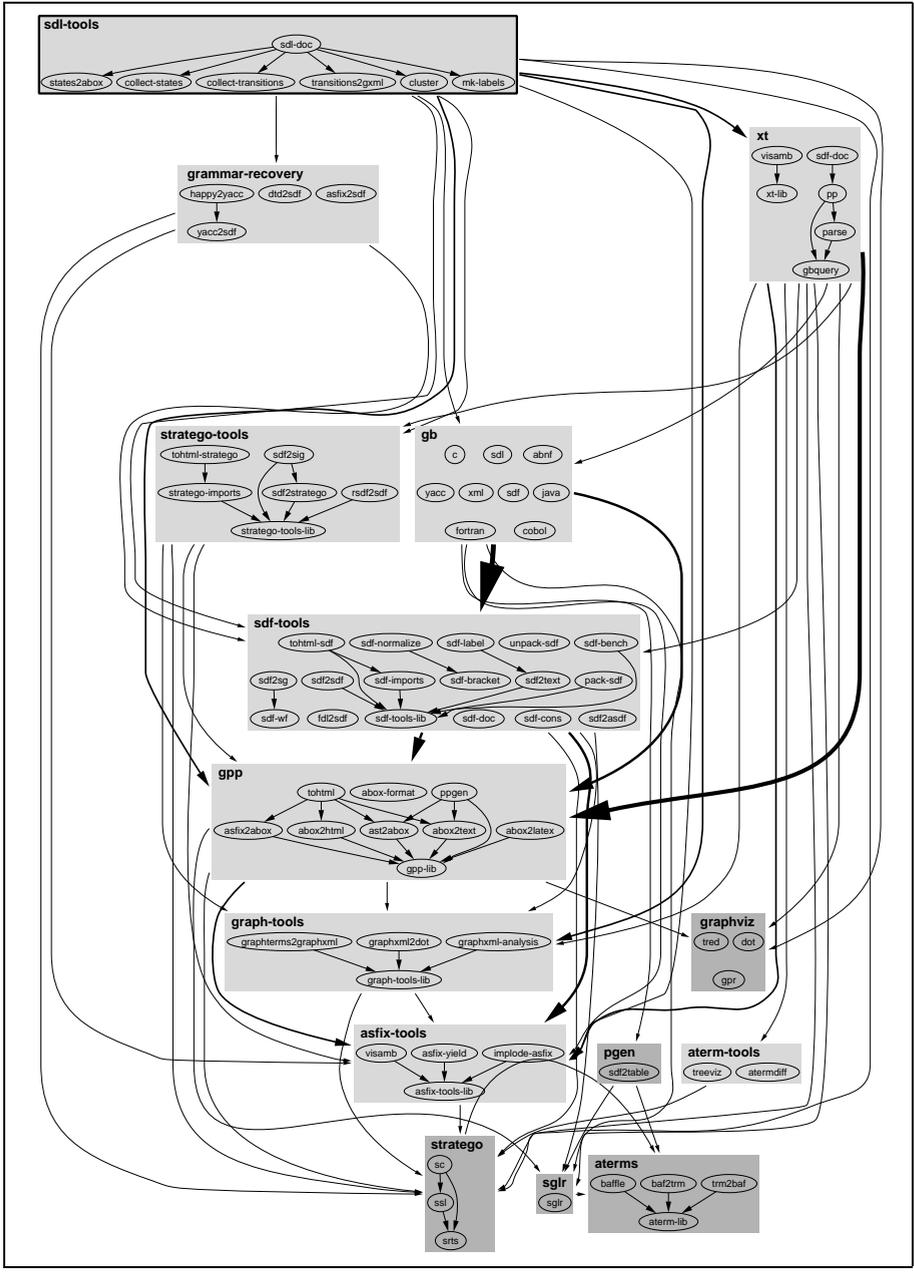
Figure 5.10 Components used by the sdl-tools package.

Source Tree Composition

A typical problem of component-based applications is their complicated construction and distribution because the internal structuring in components usually remains visible at construction and distribution-time. For example, it is not easy to deliver the SDL *documentation generator from Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages", as a unit to a customer, or to configure and build it as a unit. Consequently, each constituent component of a software system has to be separately retrieved, compiled, installed and so on.*

This chapter tackles this problem by providing techniques for automated assembly of composite software systems from their constituent source code components. This process is called Source Tree Composition and involves integration of source trees, build processes, and configuration processes. The result is a software system that hides its internal structuring in components and, consequently, can be managed as a single unit.

Application domains of source tree composition include generative programming, product line architectures, commercial off-the-shelf (COTS) software engineering, and Language-Centered Software Engineering (LCSE). The work presented in this chapter was published earlier as [81].

6.1   Introduction

The classical approach of component composition is based on pre-installed binary components (such as pre-installed libraries). This approach however, complicates software development because: (i) system building requires extra effort to configure and install the components prior to building the system itself; (ii) it yields accessibility problems to locate components and corresponding documentation [99]; (iii) it complicates the process of building self-

contained distributions from a system and all its components. Package managers (such as RPM [6]) reduce build effort but do not help much to solve the remaining problems. Furthermore, they introduce version problems when different versions of a component are used [99, 135]. They also provide restricted control over a component's configuration. All these complicating factors hamper software reuse and negatively influence granularity of reuse [112].

We argue that *source code components* (as alternative to binary components) can improve software reuse for component-based software development.[1] Source code components are source files divided in directory structures. They form the implementation of subsystems. Source code component composition yields self-contained source trees with single integrated configuration and build processes. We called this process *source tree composition*.

The literature contains many references to articles dealing with component composition on the design and execution level, and with build processes of individual components (see the related work in Section 6.9). However, techniques for composition of source trees of diverse components, developed in different organizations, in multiple languages, for the construction of systems which are to be reusable themselves and to be distributed in source, are underexposed and are the subject of this chapter.

The chapter is organized as follows. Section 6.2 motivates the need for advanced techniques to perform source tree composition. Section 6.3 describes terminology. Section 6.4 describes the process of source tree composition. Sections 6.5 and 6.6 describe abstraction mechanisms over source trees and composite software systems. Section 6.7 describes automated source tree composition. It discusses the tool `autobundle`, online package bases, and product line architectures. Section 6.8 describes experiences with source tree composition. Related work and concluding remarks are discussed in Sections 6.9 and 6.10.

6.2 Motivation

The source code components that form a software system are often tightly coupled: the implementation of all subsystems is contained in a single source tree, a central build process controls their build processes, and a central configuration process performs their static (compile-time) configuration. For example, a top-level Makefile often controls the global build process of a software system. A system is then built by recursively executing `make` [59] from the top-level Makefile for each source code component. Often, a global GNU `autoconf` [100] configuration script performs system configuration, for instance to select the compilers to use and to enable or disable debugging support.

Such tight coupling of source code components has two main advantages: (i) due to build process integration, building and configuring a system can be

---

[1]Please note that despite the advantages that source code components provide, binary components may still be mandated, for instance, to protect intellectual property.

performed easily from one central place; (ii) distributing the system as a unit is relatively easy because all source is contained in a single tree (one source tree, one product).

Unfortunately, tight coupling of source code components also has several drawbacks:

- The composition of components is inflexible. It requires adaption of the global build instructions and (possibly) its build configuration when new components are added [99]. For example, it requires adaption of a top-level Makefile to execute `make` recursively for the new component.

- Potentially reusable code does not come available for reuse outside the system because entangled build instructions and build configuration of components are not reusable [112]. For example, as a result of using `autoconf`, a component's configuration is contained in a top-level configuration script and therefore not directly available for reuse.

- Direct references into source trees of components yield unnecessary file system dependencies between components in addition to functional dependencies. Changing the file or directory structure of one component may break another.

To address these problems, the constituent source code components of a system should be isolated and be made available for reuse (*system decomposition*). After decomposition, new systems can be developed by selecting components and assembling them together (*system composition*). This process is depicted in Figure 6.1.

For system composition not only source files are required, but also all build knowledge of all constituent source code components. Therefore, we define *source tree composition* as the composition of all files, directories, and build knowledge of all reused components. To benefit from the advantages of a tightly coupled system, source tree composition should yield a self-contained source tree with central build and configuration processes, which can be distributed as a unit.

When the reuse scope of software components is restricted to a single Configuration Management (CM) [17] system, source tree composition might be easy. This is because, ideally, a CM system administrates the build knowledge of all components, their dependencies, etc., and is able to perform the composition automatically.[2]

When the reuse scope is extended to multiple projects or organizations, source tree composition becomes harder because configuration management (including build knowledge) needs to be untangled [40, 112]. Source tree composition is further complicated when third party components are reused, when the resulting system has to be reusable itself, and when it has to be

---

[2]Observe that in practice, CM systems are often confused with version management systems. The latter do not administrate knowledge suitable for source tree composition.
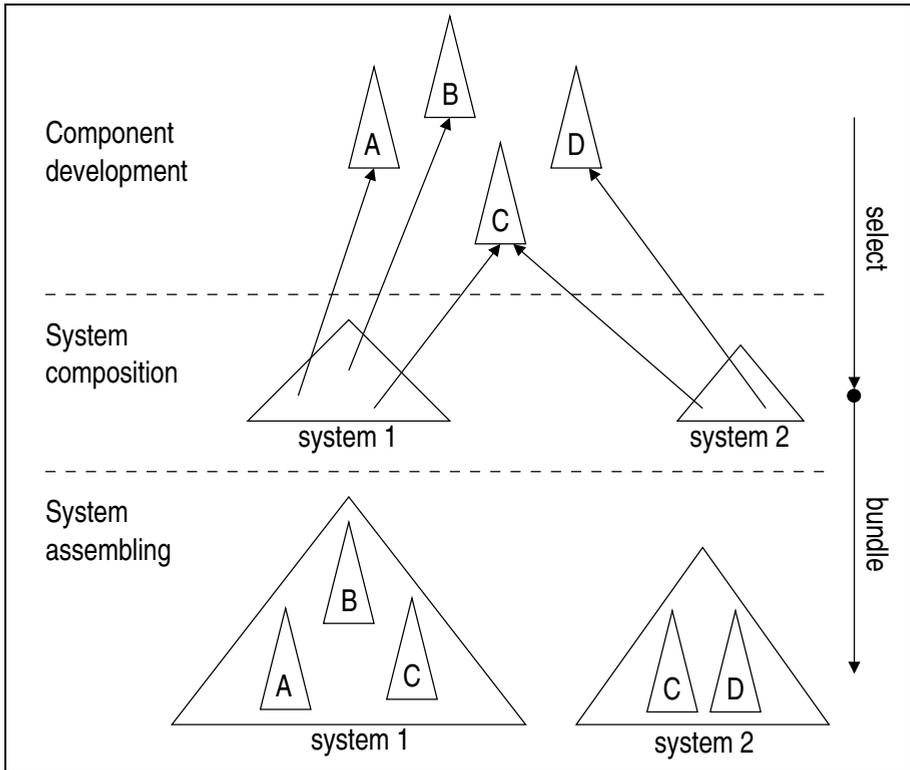
Figure 6.1 Component development, system composition, and system assembly with source code component reuse. Components are developed individually; compositions of components form systems, which are assembled to form *software bundles* (self-contained software systems).

distributed as source. This is because: i) standardization of CM systems is lacking [112, 151]; ii) control over build processes of third party components is restricted; iii) expertise on building the system and its constituent components might be unavailable.

Summarizing, to increase reuse of source code components, source tree composition should be made more generally applicable. This requires techniques to hide the decomposition of systems at distribution time, to fully integrate build processes of (third party) components, and to minimize configuration and build effort of the system. Once generally applicable, source tree composition simplifies assembling component-based software systems from implementing source code components.

Suppliers of Commercial Off-The-Shelf (COTS) source code components and of Open Source Software (OSS) components can benefit from the techniques presented in this chapter because integration of their components is

simplified, which makes them suitable for widespread use. Moreover, as we will see in Section 6.7.3 and in Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages", product line architectures, which are concerned with assembling families of related applications, can also benefit from source tree composition.

6.3 Terminology

System building is the process of deriving the targets of a software system (or software component) from source [47]. We call the set of targets (such as executables, libraries, and documentation) a *software product*, and define a *software package* as a distribution unit of a versioned software system in either binary or source form.

A system's *build process* is divided in several steps, which we call *build actions*. They constitute a system's *build interface*. A build action is defined in terms of *build instructions* which state how to fulfill the action. For example, a build process driven by `make` typically contains the build actions `all`, `install`, and `check`. The `all` action, which builds the complete software product, might be implemented as a sequence of build instructions in which an executable is derived from `C` program text by calling a compiler and a linker.

System building and system behavior can be controlled by *static configuration* [51]. Statically configurable parameters define at compile-time which parts of a system to build and how to build them. Examples of such parameters are debug support (by turning debug information on or off), and the set of drivers to include in an executable. We call the set of statically configurable parameters of a system a *configuration interface*.

We define a *source tree* as a directory hierarchy containing *all* source files of a software (sub) system. A source tree includes the sources of the system itself, files containing build instructions (such as Makefiles), and configuration files, such as `autoconf` configuration scripts.

6.4 Source tree composition

Source tree composition is the process of assembling software systems by putting source trees of reusable components together. It involves merging source trees, build processes, and configuration processes. Source tree composition yields a single source tree with centralized build and configuration processes.

The aim of source tree composition is to improve reusability of source code components. To be successful, source tree composition should meet the following two requirements:

Repeatable To benefit from any evolution of the individual components, it is essential that an old version of a component can easily be replaced by a

```
package
identification
        name=CobolSQLTrans
      version=1.0
      location=http://www.coboltrans.org
          info=http://www.coboltrans.org/doc
  description='Transformation framework for COBOL with embedded SQL'
     keywords=cobol, sql, transformation, framework
configuration interface
  layout-preserving    'Enable layout preserving transformations.'
requires
  cobol 0.5 with lang-ext=SQL
  asf 1.1 with traversals=on
  sglr 3.0
  gpp 2.0
```

Figure 6.2   An example package definition.

newer. Repeating the composition should therefore take as little effort as possible.

Invisible A source distribution of a system for non-developers should be offered as a unit (one source tree, one product), the internal structuring in source code components should not necessarily be visible. Integrating build and configuration processes of components is therefore a prerequisite.

Due to lacking standardization of build and configuration processes, these requirements are hard to satisfy. Especially when drawing on a diverse collection of software components, developed and maintained in different institutes, by different people, and implemented in different programming languages. Composition of source trees therefore often requires fine-tuning a system's build and configuration process, or even adapting the components themselves.

To improve this situation, we propose to formalize the parameters of source code packages and to hide component-specific build and configuration processes behind interfaces. A standardized *build interface* defines the build actions of a component. A *configuration interface* defines a component's configurable items. An integrated build process is formed by composing the build actions of each component sequentially. The configuration interface of a composed system is formed by merging the configuration interfaces of its constituent components.

6.5   Definition of single source trees

We propose *source code packages* as unit of reuse for source tree composition. They help to: i) easily distinguish different versions of a component and to allow them to coexist; ii) make source tree composition institute and project-independent because versioned distributions are independent of any CM system; iii) allow simultaneous development and use of source code components.

To be effectively reusable, software packages require abstractions [92]. We introduce *package definitions* as abstraction of source code packages. We developed a domain-specific language to represent them, of which an example is depicted in Figure 6.2. It defines the software package CobolSQLTrans which is intended to develop transformations for COBOL with embedded SQL.

Package definitions define the parameters of packages, which include package identification, package dependencies, and package configuration.

Package identification   The minimal information that is needed to identify a software package are its name and version number. These, as well as the URL where the package can be obtained, a short description of the package, and a list of keywords are recorded in a package's identification section (see Figure 6.2).

Package configuration   The configuration interface of a software package is defined in the configuration interface section. In Figure 6.2, the configuration interface defines a single configuration parameter and a short usage description of this parameter. With this parameter, support for layout preserving transformations in the CobolSQLTrans package can be turned on or off. Partial configuration enforced by other components and composition of configuration interfaces is discussed in Section 6.6.

Package dependencies   To support true *development with reuse*, a package definition can list the packages that it reuses in the requires section. Package definitions also allow to define a (partial) static configuration for required packages. Package dependencies are used during *package normalization* (see Section 6.6) to synthesize the complete set of packages that form a system. For example, the package of Figure 6.2 requires at least version 0.5 of the cobol package and configures it with embedded SQL. Further package requirements are the Algebraic Specification Formalism (ASF) as programming language with support for automatic term traversal, a parser (sglr), and a pretty-printer (GPP).

```
bundle
  name=CobolSQLTrans-bundle version=1.0
  configuration interface
    layout-preserving
      'Enable layout preserving transformations.'
    boxenv
      'Location of external boxenv package.'
  bundles
    package
      name=sdf version=2.1
      configuration
    package
      name=sql version=0.2
      configuration
    package
      name=cobol version=0.5
      configuration
        lang-ext=SQL
    package
      name=aterm version=1.6.3
      configuration
    package
      name=asf version=1.1
      configuration
        traversals=on
    package
      name=sglr version=3.0
      configuration
    package
      name=gpp version=2.0
      configuration
    package
      name=CobolSQLTrans version=1.0
      configuration
```

Figure 6.3   Bundle definition obtained by normalizing the package definition of Figure 6.2. This definition has been stripped due to space limitations.

6.6   Definition of composite source trees

A *software bundle* is the source tree that results from a particular source tree composition. A *bundle definition* (see Figure 6.3) defines the ingredients of a bundle, its configuration interface, and its identification. The ingredients of a bundle are defined as composition of package definitions.

A bundle definition is obtained through a process called *package normaliza-*
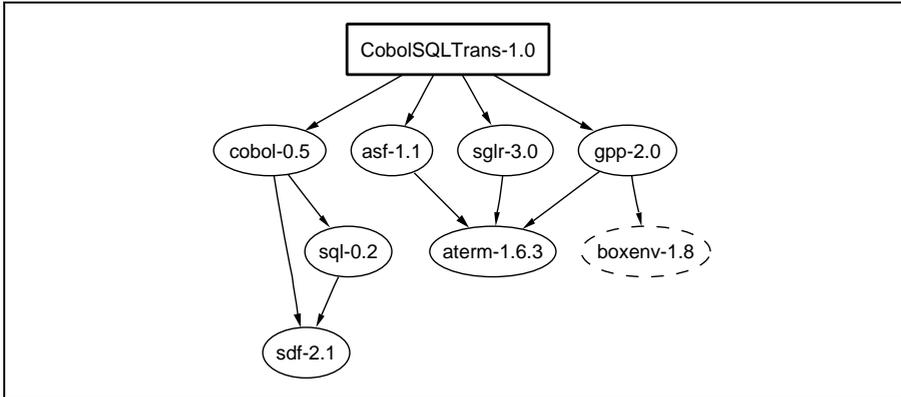
Figure 6.4   A package dependency graph for the COBOL transformation package of Figure 6.2. The dashed node denotes an unresolved package dependency.

tion which includes package dependency and version resolution, build order arrangement, configuration distribution, and bundle interface construction.

Dependency resolution   Unless otherwise specified, package normalization calculates the transitive closure of *all* required packages and collects all corresponding package definitions. The list of required packages follows directly from the bundle's package dependency graph (see Figure 6.4). For instance, during normalization of the package definition of Figure 6.2, dependency upon the aterm package is signaled and its definition is included in the bundle definition. When a package definition is missing (see the dashed node in Figure 6.4), a configuration parameter is added to the bundle's configuration interface (see below).

Version resolution   One software bundle cannot contain multiple versions of a single package. When dependency resolution signals that different versions of a package are required, the package normalization process should decide which version to bundle.

Essential for package normalization is compatibility between different versions of a package (see [149, 47, 151] for a discussion of version models). In accordance with [110], we require *backwards compatibility* to make sure that a particular version of a package can always be replaced by one of its successors. When backwards compatibility of a package cannot be satisfied, a new package (with a different name) should be created. Our tooling can be instantiated with different version schemes allowing experimenting with other (weakened) version requirements.

Build order arrangement   Package dependencies serve to define the build order of composite software systems: building a package should be delayed until

all of its required packages have been built. During package normalization, the collected package definitions are correctly ordered linearly according to a bottom-up traversal of the dependency graph. Therefore, the cobol package occurs after the sql package in the bundle definition of Figure 6.3. Circular dependencies between packages are not allowed. Such circularities correspond to bootstrapping problems and should be solved by package developers (for instance by splitting packages or by creating dedicated bootstrap packages).

Configuration propagation   Each package definition that is collected during package normalization contains a (possibly empty) set of configurable parameters, its configuration interface. Configurable parameters might get bound when the package is used by another package imposing a particular configuration. During normalization, this configuration is determined by collecting all the bindings of each package. For example, the CobolSQLTrans package of Figure 6.2 binds the configurable parameter lang-ext of the cobol package to SQL, the parameter traversals of the asf package is bound to on (see Figure 6.3). A conflicting configuration occurs when a single parameter gets bound differently. As an example, consider a software bundle that bundles packages A (which has a debug configuration switch), B, and C. A configuration conflict occurs when package B uses the debug switch of package A to turn debug support on, while package B uses it to turn debugging off. Such configuration conflicts can easily be detected during package normalization.

Bundle interface construction   The configuration interface of a bundle is formed by collecting all unbound configurable parameters of bundled packages. In addition, it is extended with parameters for unresolved package requirements and for packages that have been explicitly excluded from the package normalization process. These parameters serve to specify the installation locations of missing packages at compile-time. The configuration interface of the CobolSQLTrans package (see Figure 6.3) is formed by the layout-preserving parameter originating from the CobolSQLTrans package, and the boxenv parameter which is due to the unresolved dependency of the gpp package (see Figure 6.4).

After normalization, a bundle definition defines a software system as collection of software packages. It includes package definitions of all required packages and configuration parameters for those that are missing. Furthermore, it defines a partial configuration for packages and their build order. This information is sufficient to perform a composition of source trees. In the next section we discuss how this can be automated.

`Makefile.am`	Top-level `automake` Makefile that integrates build processes of all bundled packages.
`configure.in`	An `autoconf` configuration script to perform central configuration of all packages in a software bundle.
`pkg-list`	A list of the packages of a bundle, their versions, and download locations.
`collect`	A tool that downloads, unpacks, and integrates the packages listed in `pkg-list`.
`README`	A file that briefly describes the software bundle and its packages.
`acinclude.m4`	A file containing extensions to `autoconf` functionality to make central configuration of packages possible.

Table 6.1 Files that are contained in a generated software bundle.

6.7   Automated source tree composition

We automated source tree composition in the tool `autobundle`. In addition, we implemented tools to make package definitions available via *online package bases*. Online package bases form central meeting points for package developers and package users, and provide online package selection, bundling, and contribution via Internet. These techniques can be used to automate system assembling in product line architectures.

6.7.1   Autobundle

Package normalization and bundle generation are implemented by `autobun-dle`.[3] This tool produces a software bundle containing top-level configuration and build procedures, and a list of bundled packages with their download locations (see Table 6.1).

The generated bundle does not contain the source trees of individual packages yet, but rather the tool `collect` that can collect the packages and integrate them in the generated bundle automatically. The reason to generate an empty bundle is twofold: i) since `autobundle` typically runs on a server (see Section 6.7.2), collecting, integrating, and building distributions would reduce server performance too much. By letting the user perform these tasks, the server gets relieved significantly. ii) It protects an `autobundle` server from legal issues when copyright restrictions prohibit redistribution or bundling of packages because no software is redistributed or bundled at all.

To obtain the software packages and to build self-contained distributions, the build interface of a generated bundle contains the build actions `collect`, to download and integrate the source trees of all packages, and `bundle` to also

---

[3]See Appendix A for information about the availability of `autobundle`.

`all`	Build action to build all targets of a source code package.
`install`	Build action to install all targets.
`clean`	Build action to remove all targets and intermediate results.
`dist`	Build action to generate a source code distribution.
`check`	Build action to verify run-time behavior of the system.

Table 6.2 Build actions of the standardized build interface required by `autobundle`. In addition, a tool `configure` for static configuration is also required.

put them into a single source distribution.

The generated bundle is driven by `make` [59] and offers a standardized build interface (see Table 6.2). The build interface and corresponding build instructions are generated by `autoconf` [100] and `automake` [101]. The tool `autoconf` generates software configuration scripts and standardizes static software configuration. The tool `automake` provides a standardized set of build actions by generating Makefiles from abstract build process descriptions. Currently we require that these tools are also used by bundled packages. We used the tools because they are freely available and in widespread use. However, they are not essential for the concept of source tree composition. Essential is the availability of a standardized build interface (such as the one in Table 6.2); any build system that implements this interface would suffice. Moreover, when a build system does not implement this interface, it would not be difficult to hide the package-specific configuration and build instructions behind the standardized build interface.

After the packages are automatically collected and integrated, the top-level build and configuration processes take care of building and configuring the individual components in the correct order. The build process also provides support for generating a self-contained source distribution from the complete bundle. This hides the structuring of the system in components and allows a developer to distribute his software product as a single unit. The complete process is depicted in Figure 6.5.

6.7.2 Online package bases

Resolution of package dependencies is performed by searching for package definitions in *package repositories*. We developed tools to make such repositories browsable and searchable via Inter/Intranet, and we implemented HTML form generation for interactive package selection. The form constitutes an *online package base* and lists packages and available versions together with descriptions and keywords. The form can be filled out by selecting the packages of need. By pressing the "bundle" button, the `autobundle` server is requested to generate the desired bundle. Anyone can contribute by filling out an online *package contribution form*. After submitting this form, a package definition is
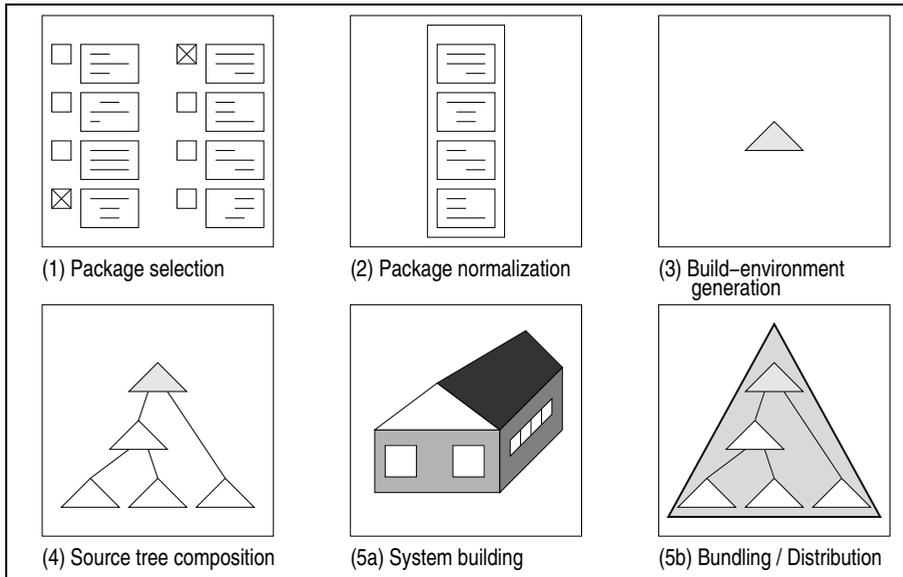
Figure 6.5 Construction and distribution of software systems with source tree composition. (1) Packages of need are selected. (2) The selected set of packages is normalized to form a bundle definition. (3) From this definition an empty software bundle is generated. (4) Required software packages are collected and integrated in the bundle, after which the system can be built (5a), or be distributed as a self-contained unit (5b).

generated and the online package base is updated. This is the only required step to make an `autoconf/automake`-based package available for reuse with `autobundle`.

Online package bases can be deployed to enable and control software reuse within a particular *reuse scope* (for instance, group, department, or company wide). They make software reuse and software dependencies explicit because a distribution policy of software components is required when source code packages form the unit of reuse.

6.7.3 Product line architectures

Online package bases allow the software engineer to easily assemble systems by selecting components of need. An assembled system is partly configured depending on the combination of components. Remaining variation points can be configured at compile-time. This approach of system assembly is related to the domain of product line architectures.

A Product Line Architecture (PLA) is a design for families of related applications; application construction (also called product instantiation [66]) is accomplished by composing reusable components [10]. The building blocks from which applications are assembled are usually abstract requirements (con-

sisting of application-oriented concepts and features). For the construction of the application, corresponding implementation components are required. To automate component assembly, *configuration knowledge* is required which maps between the *problem space* (consisting of abstract requirements) and the *solution space* (consisting of implementation components) [50].

We believe that package definitions, bundle generation, and online package bases serve implementing a PLA by automating the integration of source trees and static configuration. Integration of functionality of components still needs to be implemented in the components themselves, for instance as part of a component's build process.

Our package definition language can serve as a *configuration DSL* (Domain-Specific Language) [51]. It then serves to capture configuration knowledge and to define mappings from the problem space to the solution space. Abstract components from the problem space are distinguished from implementation components by having an empty location field in their package definition. A mapping is defined by specifying an implementation component in the requires section of an abstract package definition.

System assembling can be automated by autobundle. It normalizes a set of abstract components (features) and produces a source tree containing all corresponding implementation components and generates a (partial) configuration for them. Variation points of the assembled system can be configured statically via the generated configuration interface. An assembled system forms a unit which can easily be distributed and reused in other products.

Definitions of abstract packages can be made available via online package bases. Package bases then serve to represent application-oriented concepts and features similar to feature diagrams [86]. This makes assembling applications as easy as selecting the features of need.

Using source tree composition for product lines is further explored in Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages".

6.8 Case studies

System development   We successfully applied source tree composition to the ASF+SDF Meta-Environment [27], an integrated environment for the development of programming languages and tools, which has been developed at our research group. Source tree composition solved the following problems that we encountered in the past:

- We had difficulties in distributing the system as a unit. We were using ad-hoc methods to bundle all required components and to integrate their build processes.

- We were encountering the well-known problem of simultaneously developing and using tools. Because we did not have a distribution policy for

individual components, development and use of components were often conflicting activities.

- Most of the constituent components were generic in nature. Due to their entangling in the system's source tree however, reuse of individual components across project boundaries proved to be extremely problematic.

After we started using source tree composition techniques, reusability of our components greatly improved. This was demonstrated by the development of XT, a bundle of program transformation tools (see Chapter 3). It bundles components from the ASF+SDF Meta-Environment together with a diverse collection of components related to program transformation. Currently, XT is assembled from 25 reusable source code components developed at three different institutes.[4]

For both projects, package definitions, package normalization, and bundle generation proved to be extremely helpful for building self-contained source distributions. With these techniques, building distributions of the ASF+SDF Meta-Environment and of XT became a completely automated process. Defining the top-level component of a system (i.e., the root node in the system's package dependency graph) suffices to generate a distribution of the system.

Online Package Base To improve flexibility of component composition, we defined package definitions for all of our software packages, included them in a single package repository and made that available via Internet as the Online Package Base (see Figure 6.6).

With the Online Package Base (OPB), building source distributions of XT and of the ASF+SDF Meta-Environment becomes a dynamic process and reduces to selecting one of these packages and submitting a bundle request to the `autobundle` server. The exact contents of both distributions can be controlled for specific needs by in/excluding components, or by enforcing additional version requirements of individual components. Similarly, any composition of our components can be obtained via the OPB.

Although it was initiated to simplify and increase reuse of our own software packages, anyone can now contribute by filling out a *package contribution form*. Hence, compositions with third-party components can also be made. For example, the OPB contains several package definitions for GNU software, the graph drawing package graphviz from AT&T, and components from a number of other research institutes.

Stratego compiler Recently, the Stratego compiler [140] has been split up in reusable packages (including the Stratego run-time system). The constituting components (developed at different institutes) are bundled with `autobundle` to form a stand-alone distribution of the compiler. With `autobundle` also

---

[4]See Appendix A for information about the availability of the Online Package Base.

more fine-grained reuse of these packages is possible. An example is the distribution of a compiled Stratego program with only the Stratego run-time system. The Stratego compiler also illustrates the usefulness of *nested* bundles. Though a composite bundle, the Stratego compiler is treated as a single component by the XT bundle in which it is included.

Product line architectures   We have investigated the use of `autobundle` and online package bases in a commercial setting to transform the industrial application DOCGEN [57] into a product line architecture [52]. DOCGEN is a documentation generator which generates interactive, hyperlinked documentation about legacy systems. Documentation generation consists of generic and specific artifact extraction and visualization in a customer-specific layout. It is important that customer-specific code is not delivered to other customers (i.e., that certain packages are *not* bundled).

The variation points of DOCGEN have been examined and captured in a Feature Description Language (FDL) [56]. We are analyzing how feature selection (for instance the artifacts to document and which layout to use) can be performed via an online package base. Package definitions serve to map selected features to corresponding implementing components (such as specific extractors and visualizators). Such a feature set is normalized by `autobundle` to a bundle of software packages, which are then integrated into a single source tree that forms the intended customer-specific product. In Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages", we will further discuss the DOCGEN product line.

6.9   Related work

Many articles, for instance [39, 35, 46], address build processes and tools to perform builds. Tools and techniques are discussed to solve limitations of traditional `make` [59], such as improving dependency resolution, build performance, and support for variant builds. Composition of source trees and build processes is not addressed.

Gunter [65] discusses an abstract model of dependencies between software configuration items based on a theory of concurrent computations over a class of Petri nets. It can be used to combine build processes of various software environments.

Miller [104] motivates global definition of a system's build process to allow maximal dependency tracking and to improve build performance. However, to enable composition of components, independence of components (weak coupling) is important [149]. For source tree composition this implies independence of individual build processes and therefore contradicts the approach of [104]. Since the approach of Miller entangles all components of the system, we believe that it will hamper software reuse.

Netscape: Online Package base

File   Edit   View   Go   Communicator   Help

This is the *Online Package Base*: a collection of reusable Free Software packages developed at different institutes.

After selecting the packages you need, press the 'bundle' button below to obtain a software bundle containing the description of the packages you selected and those that are required by them. After downloading and unpacking the bundle, consult the README file in the bundle about the easy installation procedure of the complete software bundle.

If you want to exclude a package from a software bundle, select the package and choose 'exclude' as version number.

You can contribute your own software packages, by donating a package definition file. To do so, please fill out a package contribution form.

To get an impression about software reuse via the Package Base, visit the Package Dependency Graph.

| bundle | dependency graph | clear | Package search: [   ] | search |

apigen   ● 1.3   ○ exclude

Generates data types in C and Java that hide the ATerm library behind a type... ATerms following the signature in a typed manner.
keywords: *meta-environment, code generation, C, java, aterm*

asfsdf-meta   ○ 1.2   ● 1.1.1   ○ 1.1   ○ 1.0.1   ○ 1.0   ○ 0.8.1

The ASF+SDF Meta-Environment is an interactive tool suite for building language-oriented tools for program development, domain-specific languages, program analysis and software renovation.
keywords: *meta-environment, asf, sdf, dsl, language environment, aterm*

gb   ● 1.0   ○ 0.9   ○ 0.8   ○ 0.6   ○ exclude

The Grammar Base, a collection of...
keywords: *xt, SDF, grammar*

gpp   ● 2.3   ○ 2.2   ○ 2.1

GPP: the generic pretty-printer pac...
keywords: *XT, pretty-printer, html,...*

grammar-recovery   ● 0.3   ○ 0.2   ○ 0.1

Grammar Recovery: a collection of...
keywords: *xt, grammar, sdf, recove...*

graph-tools   ● 0.3   ○ 0.2   ○ 0.1

Graph Tools: a collection of tools f...
keywords: *XT, XML, graph, graphx...*

Save As... (type application/x-tar-gzip)

Filter
[/tmp/opb/opb]

Directories
[. .]

Files
[   ]

Selection
[/tmp/opb/asfsdf-meta-gb-bundle-0.1.tar.gz]

| OK | | Filter | | Cancel |

Figure 6.6   Automated source tree composition at the Online Package Base. See Appendix A for information about the availability of the Online Package Base.

This chapter addresses techniques to assemble software systems by integrating source trees of reusable components. In practice, such components are often distributed separately and their installation is required prior to building the system itself. This extra installation effort is problematic [135], even when partly automated by package managers (like RPM [6]). Although source tree composition simplifies software building, it does not make package management superfluous. The use of package managers is therefore still advocated to assist system administrators in installing (binary) distributions of assembled systems.

The work presented in this chapter has several similarities with the component model Koala [112, 110]. The Koala model has a component description language like our package definition language, and implementations and component descriptions are stored in central repositories accessible via Internet. They also emphasize the need for backward compatibility and the need to untangle build knowledge from an SCM system to make components reusable. Unlike our approach, the system is restricted to the C programming language, and merging the underlying implementations of selected components is not addressed.

In [71], a software *release management* process is discussed that documents released source code components, records and exploits dependencies amongst components, and supports location and retrieval of groups of compatible components. Their primarily focus is component release and installation, not development of composite systems and component integration as is the case in this chapter.

6.10   Concluding remarks

This chapter addresses software reuse based on source code components and on software assembly using the technique source tree composition. Source tree composition integrates source trees and build processes of individual source code components to form self-contained source trees with single integrated configuration and build processes.

Contributions   We provided an abstraction mechanism for source code packages and software bundles in the form of package and bundle definitions. By normalizing a collection of package definitions (package normalization) a composition of packages is synthesized. The tool `autobundle` implements package normalization and bundle generation. It fully automates source tree composition. Online package bases, which are automatically generated from package repositories, make package selection easy. They enable source code reuse within a particular reuse scope. Source tree composition can be deployed to automate dynamic system assembly in product line architectures.
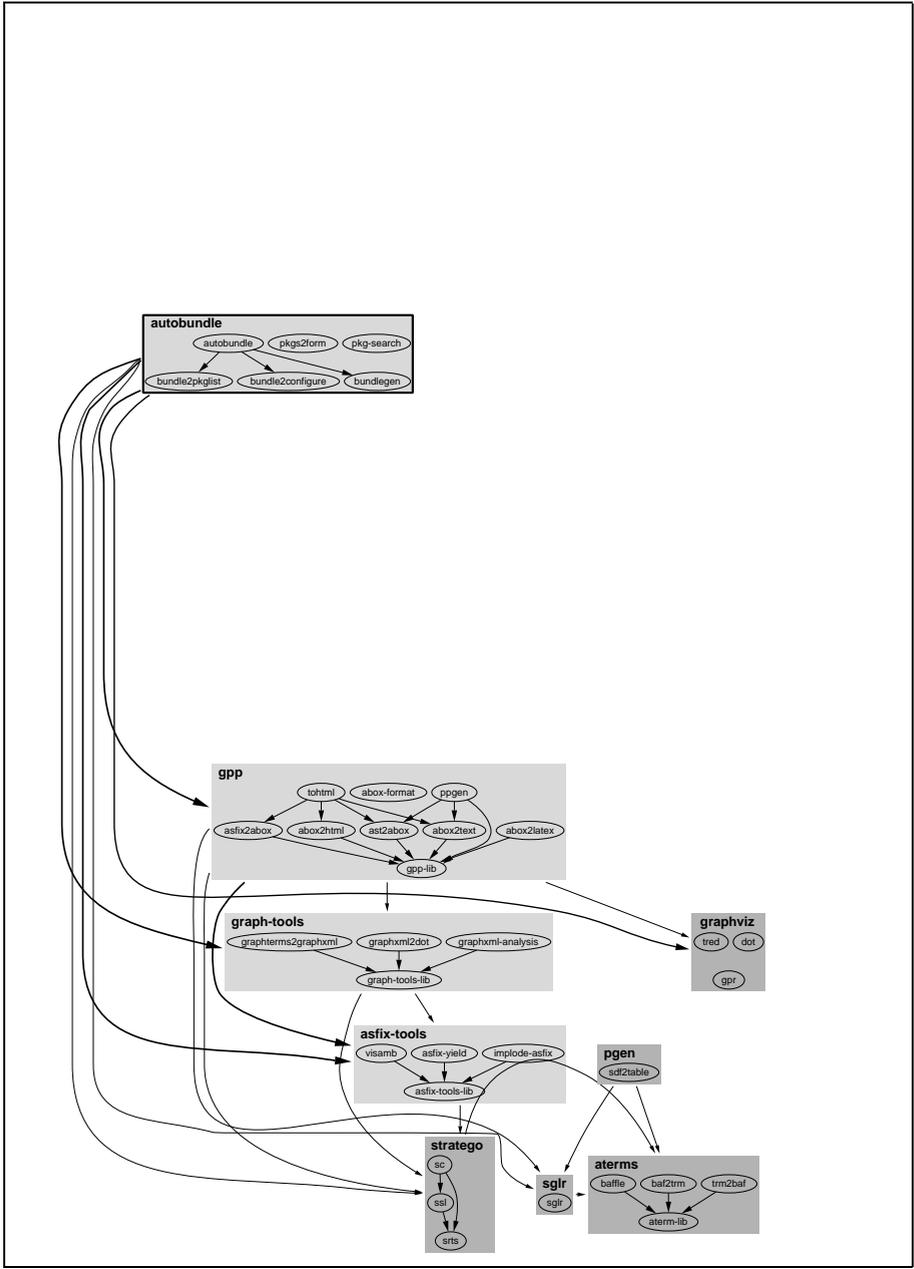
Figure 6.7 Components used for the implementation of `autobundle`.

Components and reuse The implementation of `autobundle` and the tools for generating online package bases follows the Language-Centered Software

Component	Non-transitive reuse			Transitive reuse	
	LOC	RSI	Reuse%	LOC	Reuse%
autobundle	175	0	0%	4,062	95%
bundle2configure	950	780	82%	950	82%
bundle2pkglist	576	541	93%	576	93%
bundlegen	2,361	1,505	63%	2,361	63%
pkgs2form	1,451	1,082	74%	1,451	74%
pkg-search	873	779	89%	873	89%
Totals:	**6,386**	**4,687**	**73%**	**10,273**	**83%**

Table 6.3 Reuse table for the autobundle package. The table shows that for this package, a total of 1,699 *new* lines of code had to be written.

Engineering (LCSE) model as discussed in Chapter 2, "Grammars as Contracts". For the implementation a significant amount of code is reused from the XT bundle, including generators for obtaining language-specific libraries and full-fledged components for parsing and pretty-printing.

Figure 6.7 displays the autobundle package, its constituent components, and the components it reuses (see Section 3.6 on page 42 for information about component diagrams). The autobundle package implements 6 components and reuses 10 components from 6 different packages.

Table 6.3 depicts component sizes and reuse levels of the autobundle package. The table shows that the implementation consists of approximately 6,300 lines of code, of which more than 4,600 lines are reused. This yields a reuse level between 73% and 83%. Section 3.6 justifies these numbers and describes how they are obtained by analyzing component implementations.

Future work   We depend on backwards compatibility of software packages. This requirement is hard to enforce and weakening it is an interesting topic for further research. The other requirement that we depend on now, is the use of autoconf and automake, which implement a standard configuration and build interface. We have ideas for a generic approach to hide component-specific build and configuration procedures behind standardized interfaces, but this still requires additional research.

Feature-Based Product Line Instantiation using Source-Level Packages

Chapter 6, "Source Tree Composition", discussed automated assembly and configuration of software systems from low-level, technical source code components. This chapter discusses software assembly at a higher level of abstraction using software product lines.

We discuss the construction of software products from consumer-related feature selections. We address variability management with the Feature Description Language (FDL) to capture variation points of product line architectures. We describe feature packaging using the `autobundle` *tool discussed in Chapter 6. Feature packaging covers selection, packaging, and configuration of implementation components according to feature selections. Finally, we discuss a generic approach to make instantiated (customer-specific) variability accessible in applications.*

The solutions and techniques presented in this chapter are based on our experience with the product line architecture of the commercial documentation generator DOCGEN. *The work presented in this chapter was published earlier as [52].*

7.1 Introduction

This chapter deals with three key issues in software product lines: variability management, feature packaging, and product line instantiation. It covers these topics based on our experience in the design, implementation, and deployment

of DOCGEN, a commercial product line in the area of documentation generation for legacy systems.

Like many product lines, DOCGEN started out as a single product dedicated to a particular customer. It was followed by modifications of this product for a series of subsequent customers, who all wanted a similar documentation generator, specialized to their specific needs. Gradually a kernel generator evolved, which could be instantiated to dedicated documentation requirements. DOCGEN is still evolving, and each new customer may introduce functionality that affects the kernel generator. This may involve the need for additional variation points in DOCGEN, or the introduction of functionality that is useful to a broad range of customers meriting inclusion in the standard DOCGEN setup.

The business model we use to accommodate such an evolving product line specialized to the needs of many different customers is based on subscription: customers can install a new DOCGEN version on a regular basis, which is guaranteed to be compatible with that customer's specializations. For the customer this has the advantage of being able to benefit from DOCGEN extensions; For the supplier it has the advantage of a continuous revenue flow instead of harder to predict large lump sums when selling a particular product.

The technological challenges of this business model and its corresponding evolutionary product line development are significant. In particular, product line evolution must be organized such that it can deal with many different customers. In this chapter, we cover three related issues that we experienced as very helpful to address this problem.

Our first topic is managing the variation points, which are likely to change at each release. When discussing the use of DOCGEN with a potential customer, it must be clear what the variability of the current system is. The new customer may have additional wishes which may require the creation of new variation points, extending or modifying the existing interfaces. Moreover, marketing, customer support, or the product manager may come up with ideas for new functionality, which will also affect the variability. In Section 7.3 we study the use of the *Feature Description Language* FDL described in [56] to capture such a changing variability.

The second issue we cover is managing the source code components implementing the variability. The features selected by a customer should be mapped to appropriately configured software components. In Section 7.4 we describe our approach, which emphasizes developing product line components separately, using internally released software *packages*. Assembling a product from these packages consists of merging the sources of these packages, as well as the corresponding build processes – a technique called *source tree composition* that was presented in Chapter 6. Packages can either implement a feature, or implement functionality shared by other packages.

The third topic we address is managing customer code, that is, the instantiated variability. The same feature can be implemented slightly differently for different customers. In Section 7.5 we propose an extension of the abstract

factory to achieve appropriate packaging and configuration of customer code.

In Section 7.6 we summarize our approach, contrast it with related work, and describe future directions. Before we dive into that, we provide a short introduction to the DOCGEN product line in Section 7.2.

7.2 A documentation generation product line

In this section we introduce the product line DOCGEN [57, 58], which we will use as our case study throughout the chapter. Our discussion of DOCGEN follows the format used by Bosch to present his software product line case studies [21].

7.2.1 Company background

DOCGEN is the flagship product of the Software Improvement Group (SIG), an Amsterdam, The Netherlands, based company offering solutions to businesses facing problems with the maintenance and evolution of software systems. SIG was founded in 2000, and is a spin-off of academic research in the area of reverse and reengineering conducted at CWI from 1996 to 1999. This research resulted in, amongst others, a prototype documentation generator described by [57], which was the starting point for the DOCGEN product family now offered by SIG.

7.2.2 Product family

SIG delivers a range of documentation generation products. These products vary in the source languages analyzed (such as SQL, COBOL, JCL, 4GL's, proprietary languages, and so on) as well as the way in which the derived documentation is presented.

Each DOCGEN product operates by populating a repository with a series of facts derived from legacy sources. These facts are used to derive web-based documentation for the systems analyzed. This documentation includes textual summaries, overviews, various forms of control flow graphs, architectural information, and so on. Information is available at different levels of abstraction, which are connected through hyperlinks.

DOCGEN customers have different wishes regarding the languages to be analyzed, the specific set of analyses to be performed, and the way in which the collected information should be retrieved. Thus, DOCGEN is a software product line, providing a set of reusable assets well-suited to express and implement different customized documentation generation systems.

7.2.3 Technology

At present, DOCGEN is an object-oriented application framework written in JAVA. It uses a relational database to store facts derived from legacy sources. It provides a range of core classes for analysis and presentation purposes. In order to instantiate family members, a JAVA package specific to a given customer is created, containing specializations of core classes where needed, including methods called by the DOCGEN factory for producing the actual DOCGEN instantiation. DOCGEN consists of approximately 850 JAVA classes, 750 JAVA classes generated from language definitions, 250 JAVA classes used for continuous testing, and roughly 50 shell and Perl scripts.

In addition to the key classes, DOCGEN makes use of external packages, for example for graph drawing purposes.

7.2.4 Organization

Since SIG is a relatively small company, the development team tries to work as closely together as possible. For that reason, there is no explicit separation between a core product line development team and project teams responsible for building bespoke customer products. Instead, these are roles, which are rotated throughout the entire team.

For each product instantiation, a dedicated person is assigned in order to fulfill the customer role. This person is responsible for accepting or rejecting the product derived from DOCGEN for a particular customer.

7.2.5 Process

The construction of DOCGEN is characterized by evolutionary design (the system is being developed following the principles of extreme programming [12]). DOCGEN started as a research prototype described by [57]. This prototype was not implemented as a reusable framework; instead it just produced documentation as desired by one particular customer. As the commercial interest in applications of DOCGEN grew, more and more variation points were introduced, evolving DOCGEN into a system suitable for deriving many different documentation generation systems.

With the number of customer configurations growing, it is time to rethink the way in which DOCGEN product instantiations are created, and what sort of variability the DOCGEN product line should offer.

7.3 Analyzing variability

7.3.1 Feature descriptions

To explore the variability of software product lines we use the Feature Description tion Language FDL discussed by [56]. This is essentially a textual representa-

tion for the feature diagrams of the Feature Oriented Domain Analysis method FODA [86].

A feature can be *atomic* or *composite*. We will use the convention that names of atomic features start with a lower case letter and names of composite features start with an upper case letter. Note that atomic and composite features are called features, respectively, subconcepts in [51].

An FDL definition consists of a number of *feature definitions*: a feature name followed by "`:`" and a *feature expression*. A feature expression can consist of:

- atomic features;

- composite features: named features with separate definitions;

- optional features: feature expressions followed by "?";

- mandatory features: lists of feature expressions enclosed in all();

- alternative features: lists of feature expressions enclosed in one-of();

- feature selections:[1] lists of feature expressions enclosed in more-of();

- incomplete specified feature sets of the form "...".

An FDL definition generates all possible feature configurations (also called *product instances*). Feature configurations are flat sets of features of the form all($a_1, \ldots, a_n$), where each a_i denotes an atomic feature. In [56] a series of FDL manipulations is described, to bring any FDL definition into a *disjunctive normal form* defined as:

$$\text{one-of(all}(a_{11}, \ldots, a_{1n_1}), \ldots, \text{all}(a_{m1}, \ldots, a_{mn_m}))$$

By bringing an FDL definition in disjunctive normal form, a feature expression is obtained that lists all possible configurations.

Feature combinations can be further restricted via *constraints*. We will adopt the following constraints:

- A_1 requires A_2: if feature A_1 is present, then feature A_2 should also be present;

- A_1 excludes A_2: if feature A_1 is present, then feature A_2 should not be present.

Such constraints are called *diagram constraints* since they express fixed, inherent, dependencies between features in a diagram.

---

[1]Called "or-features" in [51].

```
DocGen :
  all(Analysis, Presentation, Database)
Analysis :
  all(LanguageAnalysis, versionManagement?, subsystems?)
LanguageAnalysis :
  more-of(Cobol, jcl, sql, delphi, progress, ...)
Cobol :
  one-of(ibm-cobol, microfocus-cobol, ...)
Presentation :
  all(Localization, Interaction, MainPages, Visualizations?)
Localization :
  more-of(english, dutch)
Interaction :
  one-of(static, dynamic)
MainPages :
  more-of(ProgramPage, copybookPage, StatisticsPage, indexes, searchPage,
          subsystemPage, sourcePage, sourceDifference, ...)
ProgramPage :
  more-of(program-summary, files-used, programs-called, activation, ...)
StatisticsPage :
  one-of(statsWithHistory, statsNoHistory)
Visualizations :
  more-of(performGraph, conditionalPerformGraph, jclGraph, subsystemGraph,
          overviewGraph, ...)
Database :
  one-of(db2, oracle, mysql, ...)
```

Figure 7.1 Some of the configurable features of the DocGen product line expressed in the Feature Description Language (FDL).

7.3.2 DocGen features

A selection of the variable features of DocGen and some of their constraints are shown in Figures 7.1 and 7.2. The features listed describe the variation points in the current version of DocGen. One of the goals of constructing the FDL specification of these features is to search for alternative ways in which to organize the variable features, in order to optimize the configuration process of DocGen family members. Another goal of an FDL specification is to help (re-) structuring the implementation of product lines.

The features listed focus on just the Analysis and Presentation configuration of DocGen, as specified by the first feature definition of Figure 7.1. The Database feature in that definition will not be discussed here.

The Analysis features show how the DocGen analysis can be influenced by specifying source languages that DocGen should be able to process. The per-language parsing will actually populate a data base, which can then be used

```
%% Some constraints
subsystemPage           requires  subsystems
subsystemGraph          requires  subsystems
sourceDifference        requires  versionManagement
%% Some source language constraints
performGraph            requires  cobol
conditionalPerformgraph requires  cobol
jclGraph                requires  jcl
%% Mutually exclusive features
static                  excludes  annotations
static                  excludes  searchPage
```

Figure 7.2 Constraints on variable DocGen features.

for further analysis.

Other features are optional. For example, the feature *versionManagement* can be switched on, so that differences between documented sources can be seen over time. When a system to be documented contains subsystems which need to be taken into account, the *subsystems* feature can be set.

The Presentation features affect the way in which the facts contained in the repository are presented to DocGen end-users. As an example, the Localization feature indicates which languages are supported (English or Dutch). At compile-time, one or more supported languages can be selected; at run-time, the end-user can use a web-browser's localization scheme to actually select a language.

The Interaction feature determines the moment the HTML pages are generated. In *dynamic* interaction, a page is created whenever the end-user requests a page. This has the advantage that the pages always use the most up-to-date information from the repository and that interactive browsing is possible. In *static* mode, all pages are generated in advance. This has the advantage that no web-server is needed to inspect the data and that they can easily be viewed on a disconnected laptop. As we will see, the Interaction feature puts constraints on other presentation features.

The MainPages feature indicates the contents of the root page of the derived documentation. It is a list of standard pages that can be reused, implemented as a many-to-one association to subclasses of an abstract "Page" class. Of these, the ProgramPage consists of one or more sections.

In addition to pages, presentation includes various Visualizations. These are all optional, allowing a customer to choose to have plain (HTML) documentation (which requires less software to be installed at the client side) or graphically enhanced documentation (which requires plug-ins to be installed).

7.3.3 DOCGEN feature constraints

Figure 7.2 lists several constraints restricting the number of valid DOCGEN configurations of the features listed in Figure 7.1.

The pages that can be presented depend on the analyses that are conducted. If we want to show a *subsystemGraph*, we need to have selected *subsystems*. Some features are language-specific: a *jclGraph* can only be shown when *jcl* is one of the analyzed languages.

Last but not least, certain features are in conflict with each other. In particular, the *annotations* feature can be used to let the end-user interactively add annotations to pages, which are then stored in the repository. This is only possible in the dynamic version, and cannot be done if the Interaction is set to *static*. The same holds for the dynamic *searchPage*.

7.3.4 Evaluation

Developing and maintaining a feature description for an existing application gives a clear understanding of the variability of the application. It can be used not only during product instantiation, but also when discussing the the design of the product line. As an example, discussions about the DOCGEN feature description have resulted in the discovery of several potential inconsistencies, as well as suggestions for resolving them.

One of the problems with the use of feature descriptions is that feature dependencies can be defined in multiple ways, for instance as a composite feature definition or as a combination of a feature definition and constraints. We are still experimenting with using these different constructs in order to develop heuristics about when to use which construct.

7.4 Software assembly

7.4.1 Source tree composition[2]

Source tree composition is the process of assembling software systems by merging reusable source code components. A *source tree* is defined as a collection of source files, together with *build instructions*, divided in a directory hierarchy. We call the source tree of a particular part of a product line architecture a *source code component*. Source code components can be developed, maintained, tested, and released individually.

Source tree composition involves merging source trees, build processes, and configuration processes. It results in a single source tree with centralized build and configuration processes.

Source tree composition requires abstractions over source code components which are called *package definitions* (see Figure 7.3). A package definition

---

[2]This section is a summary of Chapter 6, "Source Tree Composition".

```
package
identification
        name=extract-programs-called
     version=3.20
    location=http://www.cwi.nl/˜mdejonge/docgen
        info=http://www.cwi.nl/˜mdejonge/docgen
  description='DocGen program-called extraction'
    keywords=cobol, extraction, extract programs called
configuration interface
   statistics   'Toggle collection of programs-called statistics (cross cutting)'
requires
   docgen-base 1.0
   extract-program 1.0
   database 1.0
   file-io 1.0 with file-extension=cob
```

Figure 7.3 Example of a *concrete* package definition for DOCGEN. The 'configuration interface' section lists configurable items together with corresponding short descriptions. The 'requires' section defines package dependencies and their configuration.

captures information about a component, such as dependencies on other components, and configuration parameters. Package definitions as the one in Figure 7.3 are called *concrete* package definitions because they correspond directly to an implementing source code component. *Abstract* package definitions, on the other hand, do not correspond to implementing source code components. They only combine existing packages and set configuration options. Abstract package definitions are distinguished from concrete package definitions by having an empty location field (see Figure 7.4).

After selecting components of need, a software *bundle* (containing all corresponding source trees) is obtained through a process called *package normalization*. This process includes package dependency and version resolution, build order arrangement, configuration distribution, and bundle interface construction.

Package definitions are stored centrally in (online) *package repositories* (see Figure 6.6 on page 113). They can be accessed by developers to search for reusable packages. They are also accessed by the tool `autobundle` (see below) to resolve package dependencies automatically when assembling product instances.

Source tree composition is automated by the tool `autobundle`. Given a set of package names, it (i) obtains their package definitions from package repositories; (ii) calculates the transitive closure of all required packages; (iii) calculates a (partial) configuration of the individual packages; (iv) generates a self-contained source tree by merging the source trees of all required packages; (v) integrates the configuration and build processes of all bundled packages.

```
package
identification
        name=programs-called
     version=1.0
     location=
          info=http://www.software-improvers.com/
  description='Cobol presentation displaying program call graph.'
    keywords=cobol, presentation, Program-Page, programs-called
configuration interface
   localization    'Selection of supported languages (cross cutting)'
   customer        'Name identifying customer-specific issues (cross cutting)'
requires
   extract-programs-called 3.20 with statistics=on
   display-programs-called 1.7 with statistics=on
   program-page 1.9
```

Figure 7.4 Example of an *abstract* package definition for the 'programs-called' feature. It forms a mapping from the problem to the solution space.

7.4.2   Source tree composition in product lines

With the help of autobundle, assembling products on a product line can become as simple as selecting the necessary packages from online package bases. By pressing the "bundle" button, autobundle is instructed to generate a self-contained customer-specific source distribution from the selected packages and those that are required by them.

By manually selecting concrete packages from an online package base, a developer maps a selection of features to a corresponding selection of implementing source code components. This is called product configuration. Manual selection of packages forms an implicit relation between features (in the problem space) and implementation (in the solution space).

This relation between problem and solution space is a many-to-many relation: a single feature can be implemented in more than one source code package; a single source code package can implement multiple features. Selecting a feature therefore may yield a configuration of a single source code package that implements multiple features (to turn the selected feature on), or it may result in bundling a collection of packages that implement the feature together.

The relation between problem and solution space can be defined more explicitly in abstract package definitions. Abstract package definitions then correspond directly to features according to the feature description of a product line architecture (see Figure 7.4). The 'requires' section of abstract packages defines dependencies upon abstract and/or concrete packages. The latter define how to map (part of) a feature to an implementation component. During

Netscape: Online Feature Base

File   Edit   View   Go   Communicator   Help

After selecting the packages you need, press the 'bundle' button below to obtain a software bundle containing the description of the packages you selected and those that are required by them. After downloading and unpacking the bundle, consult the README file in the bundle about the easy installation procedure of the complete software bundle.

| bundle | dependency graph | clear | Package search: | cobol presentation | search |

☐ **file-index**   ● 1.0   ○ exclude

DocGen Cobol presentation which displays the list of files used by a Cobol system. Part of Main–Page presentation.
keywords: *cobol, presentation, Main–Page*

■ **files-used**   ● 1.0   ○ exclude

DocGen Cobol presentation which displays the list of

Save As... (type application/x–tar–gzip)

Filter
/tmp/opb/ofb

Directories   Files
. .   [   ]

Selection
/tmp/opb/files-used-program-summary-programs-called-bundle-0.1.tar.gz

| OK | | Filter | | Cancel |

☐ **program-activation**

☐ **program-index**

keywords: *cobol, presentation, Main–Page*

■ **program-summary**   ● 1.0   ○ exclude

DocGen Cobol presentation which displays a summary of a Cobol program. Part of Program–Page presentation.
keywords: *cobol, presentation, Program–Page, summary*

■ **programs-called**   ● 1.0   ○ exclude

DocGen Cobol presentation which displays the programs that are called by a Cobol program. Part of Program–Page presentation.
keywords: *cobol, presentation, Program–Page, programs-called*

Figure 7.5   The online feature base for DOCGEN, containing customer-related features according to the feature definition of Figure 7.1. Product instances can be assembled by selecting features of need and pressing the 'bundle' button.

package normalization, all mappings are applied, yielding a collection of implementation components.

Like concrete package definitions, also abstract package definitions can be made available at online package bases. Package bases containing only abstract package definitions are called feature bases and serve to represent application-oriented concepts and features (see Figure 7.5). With online feature bases, assembling product instances reduces to selecting the features of need.

7.4.3   Source tree composition in DocGen

To benefit from source tree composition to easily assemble different product instances, the source tree of the DocGen product line needs to be split up. Different parts of the product line then become separate source code components. Every feature as described in the feature description language should be contained in a single package definition. These package definitions may depend on other features, or on core or library packages that do not implement an externally perceivable feature, but implement general functionality.

Implementing each feature as a separate package promises a clean separation of features in the source code. Whether one feature (implementation) depends on another can be easily seen in the feature description. Currently the coupling between the feature selection and the selected packages is an informal one. More experience is needed to be able to decide whether this scheme will always work.

The source code components that implement a feature or general functionality are internally released as *source code packages*, i.e., versioned distributions of source code components as discussed in Section 7.4.1. These packages are subjected to an explicit release policy. Reuse of software in different product instances is based only on released source code components. This release and reuse policy allows different versions of a component to coexist seamlessly. Furthermore, it allows developers to control when to upgrade to a new version of a component.

Apart from the packages that implement the individual features, there are several packages implementing general functionality. Of these, the docgen package implements the user interface of the software analysis side of DocGen, as well as the infrastructure to process files and read them from disk. It also implements the Application Service Provider interface where customers offer their sources over the Internet.

In order to generate the final presentation of DocGen in HTML, a package html-lib provides us with the grammar of HTML and a number of interfaces to generate files in HTML. The various graphical representations used in Doc-Gen are bundled in the package graph which knows how to present generic graphs as PDF files.

The DocGen source code components, corresponding to customer-related features, are stored in the DocGen feature base (see Figure 7.5) from which customer-specific source trees are assembled. Compilation of such assembled

source trees is performed at the Software Improvement Group to obtain customer products in binary form. The so obtained products are then packaged and delivered to our customers.

7.4.4 Evaluation

Source tree composition applied to a product line such as DOCGEN results in a number of benefits. Probably the most important one is that by using source tree composition, it is much easier to exclude functionality from the product line. This may be necessary if customers only want to use a "low budget" edition. Moreover, it can be crucial for code developed specifically for a particular customer: such code may contain essential knowledge of a customer's business, which should not be shared with other customers.

A second benefit of using packages for managing variation points is that it simplifies product instantiation. By using a feature base, features can be easily selected, resulting in the correct composition of appropriately configured packages.

Another benefit is that by putting variable features into separate packages, the source tree is split into a series of separate source code components that can be maintained individually and independently. This solves various problems involved in monolithic source trees, such as: i) Long development/test/integration cycles; ii) Limited possibilities for safe simultaneous development due to undocumented dependencies between parts of the source tree; iii) Lack of version management and release policy for individual components. Explicitly released packages having explicitly documented dependencies help to resolve these issues.

Special attention should be paid to so-called *cross cutting* features. An example is the aforementioned localization feature, which potentially affects any presentation package. Such features result in a (global) configuration switch indicating that the feature is switched on. Observe that the implementation of these features can make use of existing mechanisms to deal with cross cutting behavior, such as aspect-oriented programming [88]: the use of source tree composition does not prescribe or exclude any implementation technique.

7.5 Managing customer code

7.5.1 Customer packages

Instantiating the DOCGEN product line for a customer amounts to:

- Selecting the variable features that should be included;

- Selecting the corresponding packages and setting the appropriate configuration switches;

```
package docgen;
public class Layout
{
  ...
  String backgroundColor = "white";
  ...
}
```

Figure 7.6 Part of the default Layout class

- Writing the customer-specific code for those features that cannot be expressed as simple switches.

As the number of different customers increases, it becomes more and more important to manage such product instantiations in a controlled and predictable way. The first step is to adopt the source tree composition approach discussed in Section 7.4, and create a separate *package* for each customer. This package first of all contains customer-specific JAVA code. Moreover, it includes a package definition indicating precisely which (versions of) other DOCGEN packages it relies on, and how they should be configured.

7.5.2   Customer factories

Customer package definitions capture the package dependencies and configuration switches. In addition to this, the JAVA code implementing the packages should be organized in such a way that it can easily deal with many different variants and specializations for different customers. This involves the following challenges:

- DOCGEN core functionality must be able to create customer-specific *objects*, without becoming dependent on these;

- The overhead in instantiating DOCGEN for a new customer should be minimal;

- It must be simple to keep the existing customer-code running when new DOCGEN variation points are created.

A partial solution is to adopt the *abstract factory* design pattern in order to deal with a range of different customers in a unified way [62]. Abstract factory "provides an interface for creating families of related or dependent objects without specifying their concrete classes". The participants of this pattern include:

- An *abstract factory* interface for creating abstract products;

- Several *concrete factories*, one for each customer, for implementing the operations to create customer-specific objects;

```
package docgen.customers.greenbank;
public class Layout extends docgen.Layout
{
  String backgroundColor = "green";
}
```

Figure 7.7   The Layout class for customer Green Bank

- A range of *abstract products*, one for each type of product that needs to be extended with customer-specific behavior;

- Several *concrete products*: per abstract product there can be different concrete products for each customer;

- The *client* uses only the interfaces declared by the abstract factory and abstract products. In our case, this is the customer-independent DOCGEN kernel package.

The abstract factory solves the problem of creating customer-specific objects. Adoption of the pattern as-is to a product line, however, is problematic if there are many different customers. Each of these will require a separate concrete factory. This can typically lead to code duplication, since many of these concrete factories will be similar.

To deal with this, we propose *customer factories* as an extension of the abstract factory design pattern. Instead of creating a concrete factory for each customer, we have one customer factory which uses a customer *name* to find customer-specific classes. Reflection is then used to create an instance of the appropriate customer-specific class.

As an example, consider the class Layout in Figure 7.6, in which the default background color of the user interface is set to "white". This class represents one of the *abstract products* of the factory pattern. The specialized version for customer Greenbank is shown in Figure 7.7. The name of this class is the same, but it occurs in the specific greenbank package. Note that this is a JAVA package, which in turn can be part of an `autobundle` source code package.

Since the Layout class represents an abstract product, we offer a static getInstance factory method, which creates a layout object of the suitable type. We do this for every constructor of Layout.

As shown in Figure 7.8, this getInstance method is implemented using a static method located in a class called CustomerFactory. A key task of this method is to find the actual class that needs to be instantiated. For this, it uses the customer's name, which is read from a centralized property file. It first tries to locate the class

```
docgen.customers.<current-customer-name>.Layout
```

```
package docgen;
public class Layout
{
  ...
  public static Layout getInstance()
  {
    return (Layout)CustomerFactory.getProductInstance(
                      docgen.Layout.class, new Object[]{});
  }
  ...
}
```

Figure 7.8 Factory code for the Layout class.

If this class does not exist (e.g., because no customization is needed for this customer) the Layout class in the specified package is identified.

Once the class is determined, JAVA's reflection mechanism is used to invoke the actual constructor. For this, an array of objects representing the arguments needed for object instantiation is used. In the example, this array is empty.

7.5.3 Evaluation

The overall effect of customer packages and customer factories is that

- One package definition is created for each customer, which exactly indicates what packages are needed for this customer, and how they should be configured;

- All customer-specific JAVA code is put in a separate JAVA package, which in turn is part of the source code package of the customer;

- Adding new customers does *not* involve the creation of additional concrete factories: instead, the customer package is automatically searched for relevant class specializations;

- Turning an existing class into a variation point, to allow customer-specific overriding, is a *local* change. Instead of an adaptation to the abstract factory used by all customers, it amounts to adding the appropriate getInstance method to the variation class.

A potential problem of the use of the *customer factory* pattern is that the heavy use of reflection may involve an efficiency penalty. For DOCGEN this has proven not to be a problem. If it is, *prototype instances* for each class can be put in a hash table, which are then cloned whenever a new instance is needed. In that case, the use of reflection is limited to the construction of the hash table.

7.6 Concluding remarks

7.6.1 Contributions

In this chapter we combined three techniques to develop and build a product line architecture. We used these techniques in practice for the DOCGEN application, but they might be of general interest to build other product lines.

Feature descriptions They live at the design level of the product line and serve to explore and capture the variability of a product line. They define features, feature interactions, and feature constraints declaratively. A feature description thus defines all possible instances of a product line architecture. A product instance is defined by making a feature selection which is valid with respect to the feature description.

Feature descriptions are also helpful in understanding the variability of a product during the development of a product line architecture. For instance, when migrating an existing software system into a product line architecture, they can help to (re)structure the system into source code components as we discussed in Section 7.4.3.

To assist developers in making product instances, an interactive graphical representation of feature descriptions (for instance based on feature diagrams or Customization Decision Trees (CDT) [74]), would be of great help. Ideally, constructing product instances from feature selections should be automated. The use of `autobundle` to automatically assemble source trees (see below), is a first step in this direction. Other approaches are described in [74, 51].

Automated source tree composition At the implementation level, we propose component-based software development and structuring of applications in separate source code components. This helps to keep source trees small and manageable. Source code components can be developed, maintained, and tested separately. An explicit release policy gives great control over which version of a component to use. It also helps to prevent a system from breaking down when components are simultaneous being used and developed. To assist developers in building product instances by assembling applications from different sets of source code components, we propose automated source tree composition. The tool `autobundle` can be used for this. Needed components can be easily selected and automatically bundled via online feature bases.

Package definitions can be made to represent features of a product on a product line. By making such (abstract) packages available via online feature bases, product instantiation becomes as easy as selecting the necessary features. After selecting the features, a self-contained source tree with an integrated build and configuration process is automatically generated.

Customer configuration When two customers want the same feature, but require slightly changed functionality, we propose the notion of customer specializations. We developed a mechanism based on JAVA's reflection mechanism to manage such customer-specific functionality.

This mechanism allows an application to consist of a core set of classes, after source tree composition, that implement the features as selected for this particular product instance. Customer specificity is accomplished by specializing the classes that are deemed customer-specific based on a global customer setting. When no particular specialization is needed for a customer, the system will fall back on the default implementation. This allows us to only implement the actual differences between each customer, and allows for maximal reuse of code.

We implemented the customer-specific mechanism in JAVA. It could easily be implemented in other languages as well.

7.6.2 Related work

RSEB, the *Reuse-driven Software Engineering Business* covers many organizational and technical issues of software product lines [73]. They emphasize an iterative process, in which an application family evolves. *Variation points* are distinguished both at the use case and at the source component level. Components are grouped into *component systems*, which are similar to our *abstract packages*. In certain cases component systems implement the *facade* design pattern, which corresponds to a facade package in our setting containing just the code to provide an integrated interface to the constituent packages.

FeatuRSEB [64] is an extension of RSEB with an explicit domain analysis phase based on FODA [86]. The feature model is used as a *catalog* of feature commonality and variability. Moreover, it acts as *configuration roadmap* providing an understanding of what can be combined, selected, and customized in a system.

Generative programming aims at automating the mapping from feature combinations to implementation components through the use of generator technology [51], such as C++ template meta-programming or GenVoca [10]. They emphasize features that "cross cut" the existing modularization, affecting many different components. Source tree composition could be used to steer such generative compositions, making use of partially shared configuration interfaces for constituent components.

Customization Decision Trees are an extension of feature diagrams proposed by Jarzabek *et al.* [74]. Features can be annotated with *scripts* specifying the architecture modifications needed to realizing the variant in question. In our setting, this could correspond to annotating FDL descriptions with package names implementing the given features.

Bosch analyzes the use of object-oriented frameworks as building blocks for implementing software product lines [21]. One of his observations is that industrial-strength component reuse is almost always realized at the source

code level. "Components are primarily developed internally and include functionality relevant for the products or applications in which it is used. Externally developed components are generally subject to considerable (source code) adaptation to match, e.g., product line architecture requirements" [21, p. 240]. *Source tree composition* as proposed in this chapter provides the support required to deal with this product line issue in a systematic and controlled way.

In another paper, Bosch *et al.* list a number of problems related to product instantiation [22]. They recognize that it is hard to exclude component features. Our proposed solution is to address this by focusing on source-level component integration. Moreover, they observe that the initialization code is scattered and hidden. Our approach addresses this problem by putting all initialization code in the abstract factory, so that concrete factories can refine this as needed. Finally, they note the importance of design patterns, including the abstract factory pattern for product instantiation.

The use of design patterns in software product lines is discussed by Sharp and Roll [129]. This chapter deals with one design pattern in full detail, and proposes an extension of the abstract factory pattern for the case in which there are many different customers and product instantiations.

The abstract factory is also discussed in detail by Vlissides, who proposes *pluggable factories* as an alternative [144, 145]. The pluggable factory relies on the *prototype* pattern (creating an object by copying a prototypical instance) in order to modify the behavior of abstract factories dynamically. It is suitable when many different, but similar, concrete factories are needed.

Anastasopoulos and Gacek discuss various techniques for implementing variability, such as delegation, property files, reflection and design patterns [1]. Our abstract factory proposal can be used for any of their techniques, and addresses the issue of *packaging* the variability in the most suitable way.

The use of *attributed* features to describe configured and versioned sets of components is covered by Zeller and Snelting [151]. They deal with configuration management only: an interesting area of future research is to integrate their feature logic with the feature descriptions of FODA and FDL.

PART III

Epilogue

Conclusions

The objective of this thesis was to develop an architecture for effective software reuse where components can be developed by different people at different institutes, and be integrated easily in composite software systems. This objective posed a number of questions about reuse techniques concerning abstraction, composition, and granularity. These questions were formulated and motivated in Section 1.5 on page 10. Section 1.6 on page 12 gave a brief overview of the research topics covered in the subsequent chapters of this thesis. In this concluding chapter, we will reflect on the research questions, summarize the reuse techniques that we developed, and draw some conclusions. Furthermore, we will discuss the effectiveness of our architecture for software reuse.

8.1   Abstraction

> **Question 1**
>
> How can an effective reuse practice in the domain of language processing be established?

To answer this question, we developed an architecture for component-based software development in Chapter 2–4, and tested its effectiveness in Chapter 5.

In Chapter 2, "Grammars as Contracts", we developed the model "Language-Centered Software Engineering" (LCSE) for component-based software development in the domain of language processing. Components in this model are stand-alone programs that can be connected via the standard exchange

Technique (chapter)	Truism			
	I	*II*	*III*	*IV*
Grammar Base (2)		✓	✓	✓
Library Generation (2)	✓	✓		
Program Generation (2)	✓	✓		
Generic Tools (2)	✓	✓		
Grammars as contracts (2)		✓	✓	
Standardized exchange formats (2)		✓		
Abstract from concrete syntax (2)		✓		
XT bundle (3)	✓	✓	✓	✓
Generic pretty-printing (4)	✓	✓		
Source tree composition (6)	✓			
Build interfaces (6)		✓	✓	
Configuration interfaces (6)		✓	✓	
Package definitions (6)	✓		✓	
Online package base (6)	✓	✓	✓	✓
Feature definition language (7)	✓			
Customer factories (7)		✓		
Online feature base (7)	✓	✓	✓	✓

I	A reuse technique must reduce the cognitive distance.
II	Reusing an artifact must be easier than developing it.
III	To select an artifact you must know what it does.
IV	Finding an artifact should be fast.

Figure 8.1 Summary of the reuse techniques that have been discussed in this thesis, together with the chapter were they have been introduced, and the reuse truisms that are satisfied by them (see Section 1.1 on page 2).

format ATERMS. The model is language-independent and allows easy integration of third-party components. Applications can be constructed from components that are reused as-is, or from components that are partly or completely generated using library and program generators. Currently, library generation support is provided for the programming languages C, HASKELL, JAVA, and STRATEGO.

Grammars play a central role in this model and serve as contracts between components. They also drive generators in order to produce compositional components that operate on uniform data structures. Grammars are stored in the Grammar Base, which is a central access point for reusable, open source language definitions. The grammar base functions as a repository of contracts, as a standard reference for language definitions, and as a starting point for application and component development.

Typical abstractions in the domain of language processing are parsers, compilers, tree transformers, and pretty-printers. XT bundles implementations for

these abstractions together with a collection of library and program generators. Chapter 3, "XT: a Bundle of Program Transformation Tools", motivates its development as to form an open framework for component-based transformation tool development, which is flexible and extendible.

The use of LCSE and of XT in practice was discussed in Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages". We demonstrated that the development time of language applications was decreased thanks to effective reuse of generators and generic language components from the XT bundle. The effectiveness of software reuse was demonstrated by a reuse level between 88% en 97%. In addition to Chapter 5, we used LCSE throughout this thesis for all software development activities. Each of these chapters concludes with a discussion of reuse statistics. Section 8.4 summarizes these results.

The individual reuse techniques that we developed in Chapters 2–5 satisfy several reuse truisms. Table 8.1 contains a summary of these techniques and indicates which reuse truisms are satisfied by them. These techniques are combined in the XT bundle. XT therefore satisfies all four truisms and can be used to establish an effective reuse practice in the domain of language processing (see Section 8.4).

8.2 Composition

> **Question 2a**
>
> How can the compositionality of components be improved and the composition process be automated?

To answer this question, we developed techniques for functional composition, source composition, and feature composition.

The composition of functional components was discussed in Chapter 2, "Grammars as Contracts". We discussed the use of grammars as contracts between language tool components and explained that the compositionality of components can be improved with centralized grammar management. We discussed meta-tooling that generates library code for a variety of programming languages from concrete and abstract syntax definitions. Thanks to centrally managed grammars, generated library code is guaranteed to operate on uniform structured trees. In combination with the ATERMS format for representing and exchanging trees, components that are constructed with these libraries can easily be connected. The use of ATERMS as exchange format makes our architecture open and language-independent. It allows composition of components from arbitrary origin, implemented in different programming languages. Table 8.1 summarizes the techniques introduced in Chapter 2 and the reuse truisms that are satisfied by them. These techniques are combined in the XT bundle, which forms an architecture that satisfies all reuse truisms. Automated

composition of functional components was only briefly addressed in Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages" and is subject of ongoing research.

Composition of source components was discussed in Chapter 6, "Source Tree Composition". We discussed abstractions for source trees and synthesis of composite trees using source tree composition. We introduced build and configuration interfaces as mechanisms to make source components compositional. They serve to integrate build processes as well as configuration processes of all the source components that constitute a software system. Source tree composition is automated in the tool `autobundle`. Online package bases make source component composition as easy as selecting the components of need. The reuse truisms that are satisfied by the techniques presented in Chapter 6 are summarized in Table 8.1. The table indicates that online package bases satisfy all truisms. As we will see below, they form a successful technique for software reuse.

Composition of features was discussed in Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages". This chapter focused on developing product lines and on automating the assembly process of product instances. To that end, we used FDL to capture configuration knowledge and to define the features of a product line. We proposed an explicit mapping from features to source components using abstract package definitions. Thanks to this mapping, a product (defined as a composition of features) can be assembled with source tree composition and the tool `autobundle` can be used to automate this process. Product assembly is further simplified by storing features in online package bases. Assembling a product then involves selecting the features of the product and pressing a button to start the assembly process. To allow behavioral adaptations to product instances according to customer-specific needs, we presented customer factories. This is a flexible mechanism for component configuration that does not put restrictions on component compositionality. Table 8.1 summarizes the reuse techniques presented in Chapter 7 and the reuse truisms that are satisfied by them. Like online package bases, online feature bases satisfy all truisms and are a powerful means for software reuse.

Question 2b

How can project and institute-specific dependencies of software components be removed in order to promote collaborative software development?

In Chapter 6, "Source Tree Composition", we proposed software reuse based on source packages. A source package is a distribution unit of a source code component that is independent of a CM system. To deal with variation over time (which is a major task of CM systems), source packages are subject to explicit release and version management. Obviously, implicit dependencies on locally

installed software are not allowed because source packages are intended for distribution. Source packages thus restrict institute-specificity of source components because they are independent of a CM system and because component developers are encouraged to drop dependencies on local installed software.

Build and configuration interfaces were proposed to standardize the build and configuration processes of source packages. They make collaborative software development easier because the software construction process becomes uniform, and because build processes as well as configuration processes of different components can easily be integrated.

Package definitions, which are abstractions for source packages, provide information about source components. This information helps to reduce the cognitive distance and to improve the understanding of components. Online package bases serve to make components widely available, and to easily find and retrieve components of need. Anybody within a reuse scope can contribute to the collaborative software development process by filling out a package contribution form at an online package base.

As part of our research, we initiated the Online Package Base (see Appendix A), which forms a central meeting point for developers and users of source packages. It provides online package selection, bundling, and contribution via Internet. Anyone can contribute additional source packages by filling out a *package contribution form*. As of this writing, the Online Package Base contains 274 packages, corresponding to 66 source code components in different variants (versions), developed at 8 institutes.

8.3   Granularity

> **Question 3**
>
> Can the conflicting goals of many, small components (fine-grained reuse) and large-scale components (high payoff and low cognitive distance) be combined?

In Chapter 6, "Source Tree Composition", we discussed a technique to assemble composite source trees from individual source code components. Assembling composite source trees involves merging source files and directories, as well as integrating build and configuration processes. The result is a single source tree with a single integrated build and configuration process.

Build and configuration interfaces were introduced to improve compositionality of source code components by making configuration and construction uniform activities. We introduced package definitions as abstractions for source code components. They capture information about components, including dependencies upon other source code components.

With build interfaces, configuration interfaces, and package definitions, coarse-grained components can easily be splitted up in smaller source code components. A package definition then serves to define a composition of smaller components. Build and configuration interfaces ensure that build and configuration processes of these components can easily and automatically be integrated.

With source tree composition, these fine-grained components can also be used in alternative compositions. Furthermore, composite components can function as building blocks themselves to form even larger components. Thus, source tree composition allows the construction of components of varying granularity.

The ability to construct components of different granularity promotes fine-grained software reuse because reusable software can be made available in small source code components. Additionally, high payoff and low cognitive distance can be achieved by making different component compositions, forming coarse-grained, domain-specific components. Package normalization ensures that common components in component compositions are always shared.

Since source tree composition is automated, the internal structuring of composite components is of no importance for component users. The `autobundle` tool takes care of obtaining and integrating the fine-grained components that constitute the intended software system. Consequently, source tree composition is almost invisible for users of composite systems and components, and the overhead is relative small.

8.4 Components and reuse

Figure 8.2 contains a complete picture of the packages developed in Chapters 3, 4, 5, and 6, their constituent components, and the component reuse relations (see Section 3.6 for information about component diagrams). The picture shows 16 packages, including 4 third-party packages, and 89 tool components.

Table 8.1 summarizes sizes and reuse levels for all the light-grey colored packages. This table shows that the complete implementation of the packages consists of approximately 77,100 lines of code, of which more than 61,700 lines are reused. Thus, the total implementation of the packages discussed in this thesis consists of 15,400 LOC. If we take the explosion factor of 1.48 into account (as discussed in Section 3.6), we end up with a total of 10,400 lines of real-written Stratego code. This yields a reuse level between 80% and 91%. Section 3.6 on page 42 justifies these numbers and describes how they are obtained by analyzing component implementations.

Recall from Section 3.6 that these numbers only depict reuse levels for components implemented in Stratego. Reuse of third-party components (contained in dark-grey boxes in Figure 8.2), implemented in other programming languages (such as the parser `sglr`, or the ATERMS library), is not depicted.
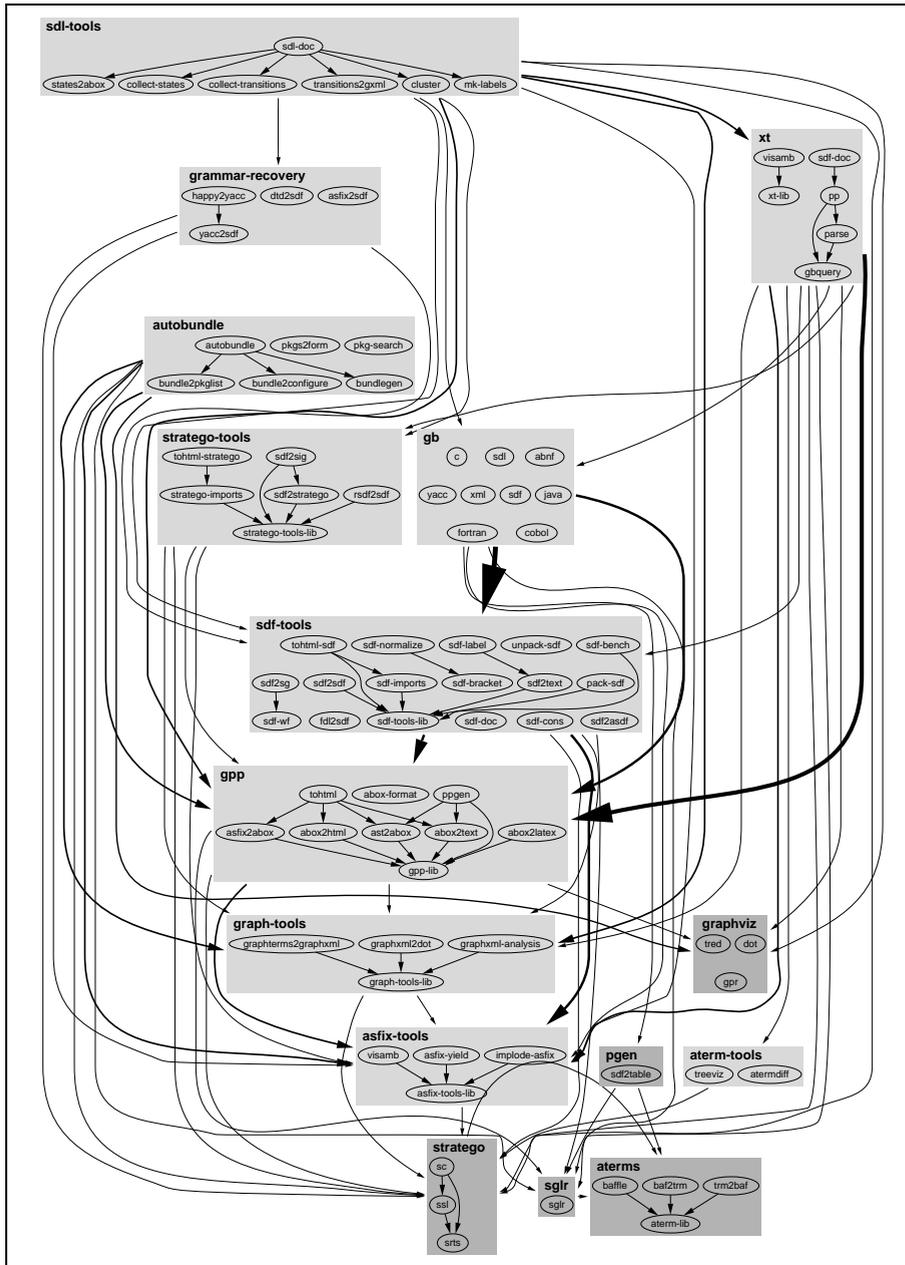
Figure 8.2   This picture shows the source code components and corresponding reuse relations for all the applications that have been discussed in this thesis.

Component	Non-transitive reuse			Transitive reuse	
	LOC	RSI	Reuse%	LOC	Reuse%
`aterm-tools`	3,616	2,999	82%	3,616	82%
`autobundle`	6,386	4,687	73%	10,273	83%
`gpp`	12,178	8,408	69%	26,349	85%
`grammar-recovery`	7,871	6,382	81%	7,871	81%
`graph-tools`	3,002	2,646	88%	3,002	88%
`sdf-tools`	23,700	20,601	86%	61,186	94%
`stratego-tools`	8,667	7,091	81%	8,667	81%
`xt`	243	0	0%	29,445	99%
`asfix-tools`	6,442	4,433	68%	6,442	68%
`sdl-tools`	5,043	4,487	88%	22,819	97%
Totals:	**77,148**	**61,734**	**80%**	**179,670**	**91%**

Table 8.1 Reuse table for the packages discussed in this thesis. The table shows that for these packages, a total of 15,414 *new* lines of code had to be written.

Consequently, software reuse is even better than the table suggests since these components are extensively used as well.

Experience reports about software reuse are discussed, amongst others, in [118] and [14]. The first contains a summary of published industrial experiences about the benefits of software reuse and reports reuse levels between 17% and 90% (55% on average). The second is concerned with an 8 year research project performed at AT&T and reports a reuse level of 85% on average. Thus, with a reuse level between 80% and 91% on average (see Table 8.1), our techniques can easily compete with the most successful ones discussed in these reports.

Contributed Software Packages

In this thesis we reported on the development of several software packages that implement the presented ideas and techniques for software reuse and collaborative software development. Most of them are freely available and distributed as open source. This section contains a summary of these packages which can all be downloaded from the Online Package Base (see below).

Online Package Base   The packages listed below, as well as tens of other packages from several different institutes, are available from the Online Package Base, which is located at:

```
http://www.program-transformation.org/package-base
```

The Online Package Base is also available at:

```
http://www.cwi.nl/~mdejonge/package-base
```

To obtain software from the Online Package Base you first select one or more of the packages, then you press the "bundle" button. This will generate a self-contained software bundle for you with the packages you selected and those that are required by them. In order to build the software bundle, the README file that is contained in the generated bundle should be consulted for instructions about the easy three-step installation procedure. The Online Package Base was discussed in Chapter 6, "Source Tree Composition".

GPP   This is a collection of tools for generic pretty-printing. The package contains a pretty-printer generator, and format engines operating on parse-trees and abstract syntax-trees, supporting different output formats. GPP was discussed in Chapter 4, "Pretty-Printing for Software Reengineering".

XT   This is a bundle of tools for building program transformations. It bundles several components from the ASF+SDF Meta-Environment [27], the generic pretty-printer GPP, the transformational programming language Stratego, and various library generators, and transformation components (see Chapter 3, "XT: a Bundle of Program Transformation Tools", for a more complete overview of its ingredients). Obtaining and installing XT is simple because binary and source distributions are available in several formats. XT distributions and documentation are available at the XT web-site:

```
http://www.program-transformation.org/xt
```

The maintainers of XT are Merijn de Jonge, Eelco Visser, and Joost Visser.

Autobundle   This software package contains tools for automated source tree composition and for managing online package bases. It includes the `auto-bundle` tool for generating self-contained software bundles, as well as tools for querying package repositories, for initiating online package bases, for web-site generation, and for handling user requests via a CGI interface. `Autobundle` was discussed in Chapter 6, "Source Tree Composition".

GB   The Grammar Base is a collection of open source language definitions in the syntax definition formalism SDF. They can be used to drive generators from the XT bundle and as contracts between language tool components for LCSE. It is available online at:[1]

```
http://www.program-transformation.org/gb
```

From this site, you can browse and download individual grammars. The complete collection of grammars is also available at the Online Package Base (see below). The Grammar Base was discussed in Chapter 2, "Grammars as Contracts".

Many people have contributed to the Grammar Base. Its maintainers are Merijn de Jonge, Eelco Visser, and Joost Visser.

---

[1]Also available at `http://www.cwi.nl/~mdejonge/gb`

Additional Software Packages

Apart from the software packages that we developed during our research, we also list a selection of additional software packages that we made use of. All these packages are open source and, in addition to the indicated locations, they are also available at the Online Package Base. From there, they can easily be retrieved and bundled with other software packages.

The ASF + SDF Meta-Environment   The ASF + SDF Meta-Environment is an interactive development environment for language prototyping and for developing program transformations. It supports combined definition of syntax and semantical aspects of (programming) languages using the syntax definition formalism SDF and the term rewriting language ASF, respectively. It as available from:

```
http://www.cwi.nl/projects/MetaEnv
```

The ATerm Library   The ATerm format is used by all our tooling as standard exchange format and is the standard data format of the Stratego programming language. The ATerm Library is therefore an important component in our Language-Centered Software Engineering model. It includes the ATerm API, which is used by for constructing and inspecting ATerms, as well as, a collection of tools for ATerm processing. The ATerm library is available from:

```
http://www.cwi.nl/projects/MetaEnv/aterm
```

Autoconf   Autoconf is a tool for generating configuration scripts for software packages. These configuration scripts perform system checks and offer switches to activate or configure parts of a package. The generated configuration scripts provide a standardized configuration interface, which `autobundle` uses for the composition of configuration processes. Autoconf is available at:

```
http://www.gnu.org/directory/GNU/autoconf.html
```

Automake   Automake is a tool for generating Makefiles from high-level Makefile templates. The generated Makefiles provide a standardized build interface, which `autobundle` uses for the composition of build processes. Automake is available at:

```
http://www.gnu.org/directory/GNU/automake.html
```

Graphviz   This package provides a collection of tools for manipulating graph structures and generating graph layouts. The tools operate on the graph drawing language DOT. The package is available at:

```
http://www.research.att.com/sw/tools/graphviz
http://www.graphviz.org
```

SDF parse table generator and generalized LR parser   The parse table generator `pgen` and the scannerless generalized LR parser `sglr` that consumes these tables are the primary tools that support the syntax definition formalism SDF. They are available from:

```
http://www.cwi.nl/projects/MetaEnv/pgen
http://www.cwi.nl/projects/MetaEnv/sglr
```

Stratego   Stratego is the primary programming language that we used for the implementation of the tools presented in this thesis. It is a modular language for the specification of fully automatic program transformation systems based on the paradigm of rewriting strategies. The Stratego distribution is available at:

```
http://www.stratego-language.org
```

Bibliography

[1] M. Anastasopoulos and C. Gacek. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 109–117. ACM, 2001. SIGSOFT Software Engineering Notes 26(3).

[2] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[3] L. Augusteijn. The elegant compiler generator system. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, September 1990.

[4] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[5] O. S. Bagge, M. Haveraaen, and E. Visser. CodeBoost: A framework for the transformation of C++ programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001.

[6] E. C. Bailey. *Maximum RPM*. Red Hat Software, Inc., 1997.

[7] S. Bailliez et al. *Apache Ant 1.5 Manual*. Apache Software Foundation, 1.5 edition, 2002. Available at `http://jakarta.apache.org/ant/manual/`.

[8] D. Batory and B. J. Geraci. Validating component compositions in software system generators. In M. Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*, pages 72–81. IEEE Computer Society Press, 1996.

[9] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 62–87, 1997.

[10] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–214, 2002.

[11] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[12] K. Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 1999.

[13] M. Becker and J. Diaz-Herrera. Creating Domain Specific Libraries: A Methodology and Design Guidelines. In *Proceedings of the Third International Conference on Software Reuse*, pages 158–168, 1994.

[14] D. Belanger and B. Krishnamurthy. Practical software reuse: An interim report. In W. Frakes, editor, *Proceedings: 3rd International Conference on Software Reuse*, pages 53–63. IEEE Computer Society Press, 1994.

[15] J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[16] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.

[17] R. H. Berlack. *Software Configuration Management*. Wiley and Sons, New York, 1991.

[18] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In W. Frakes, editor, *Proceedings: 3rd International Conference on Software Reuse*, pages 102–109. IEEE Computer Society Press, 1994.

[19] T. J. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions. In T. J. Biggerstaff and C. Richter, editors, *Software Reusability*, volume I — Concepts and Models, chapter 1, pages 1–17. ACM press, 1989.

[20] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 1998.

[21] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[22] J. Bosch and M. Högström. Product instantiation in software product lines: A case study. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, volume 2177 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag, 2000.

[23] R. J. Boulton. SYN: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical report, Computer laboratory, University of Cambridge, 1996.

[24] D. Box. *Essential COM*. Addison-Wesley, 1998.

[25] C. O. Braga, A. von Staa, and J. C. S. P. Leitte. Documentu: a flexible architecture for documentation production based on a reverse-engineering strategy. *Journal of Software Maintenance: Research and Practice*, 10(4):279–303, 1998.

[26] M. G. J. van den Brand. Prettyprinting without losing comments. Technical Report P9315, Programming Research Group, University of Amsterdam, 1993.

[27] M. G. J. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In *Compiler Construction 2001 (CC 2001)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, April 2001.

[28] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[29] M. G. J. van den Brand, P. Klint, and J. Vinju. Term rewriting with type-safe traversal functions. In B. Gramlich and S. Lucas, editors, *Second International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[30] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153. IEEE Computer Society Press, 1997.

[31] M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings*

of the sixth International Workshop on Program Comprehension, pages 108–117. IEEE Computer Society Press, 1998.

[32] M. G. J. van den Brand and J. Vinju. Rewriting with Layout. In N. Derschowitz and C. Kirchner, editors, *First International Workshop on Rule-Based Programming (RULE2000)*, 2000.

[33] 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.

[34] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, 1998.

[35] P. Brereton and P. Singleton. Deductive software building. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, volume 1005 of *Lecture Notes in Computer Science*, pages 81–87. Springer-Verlag, October 1995.

[36] H. H. Britton, R. A. Parker, and D. L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on Software engineering*, pages 195–204, 1981.

[37] A. W. Brown and G. Booch. Reusing open-source software and practices: The impact of open-source on commercial vendors. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 123–136. Springer-Verlag, 2002.

[38] P. Brown. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3):363–392, 1991.

[39] J. Buffenbarger and K. Gruel. A language for software subsystem composition. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE Computer Society Press, 2001.

[40] M. Cagan and A. Wright. Untangling configuration management. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, volume 1005 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, 1995.

[41] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, University of Colorado, April 1998.

[42] S. Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 285–299, October 1995.

[43] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[44] J. C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, July 1988.

[45] P. Clements and L. M. Nothrop et al. A framework for software product line practice. Technical Report from the Product Line System Program, Version 2.0, Software Engineering Institute, Pittsbutg, PA, July 1999.

[46] G. M. Clemm. The Odin system. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, volume 1005 of *Lecture Notes in Computer Science*, pages 241–2262. Springer-Verlag, October 1995.

[47] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[48] R. Cruickshank, H. Felber, J. Gaffney, and R. Werling. Software measurement guidebook. Technical Report SPC-91060-CMC, Software Productivity Consortium, August 1994.

[49] P. Csurgay. Prototyping framework for SDL with evolving semantics. In J. Wu, S. T. Chanson, and Q. Gao, editors, *proceedings of the IFIP TC6 WG6.1 FORTE/PSTV'99*, Formal Methods for Protocol Engineering and Distributed Systems. Kluwer Academic Publishers, 1999.

[50] K. Czarnecki and U. W. Eisenecker. Components and generative programming. In O. Nierstrasz and M. Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer-Verlag / ACM Press, 1999.

[51] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley, 2000.

[52] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In G. J. Chastek, editor, *Proceedings: Second Software Product Line Conference (SPLC2)*, number 2379 in Lecture Notes in Computer Science, pages 217–234. Springer-Verlag, August 2002.

[53] A. van Deursen and J. Visser. Building program understanding tools using visitor combinators. In *Proceedings 10th Int. Workshop on Program Comprehension, IWPC 2002*, pages 137–146. IEEE Computer Society Press, 2002.

[54] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[55] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[56] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[57] A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.

[58] Automatic Documentation Generation; White Paper. Software Improvement Group, 2001. Available at `http://www.software-improvers.com`.

[59] S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(3):255–265, March 1979.

[60] Free Software Foundation. *GNU General Public License*. Available at `http://www.fsf.org/copyleft/gpl.html`.

[61] P. Freeman. Reusable software engineering: Concepts and research directions. In T. Biggerstaff and T. E. Cheatham, Jr., editors, *Proceedings of the ITT Workshop on Reusability in Programming*, pages 129–137. ITT, 1983.

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[63] E. R. Gansner, E. Koutsofios, S. North, and K-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[64] M. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85. IEEE Computer Society Press, 1998.

[65] C. A. Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology*, 9(1):94–131, January 2000.

[66] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In R. Kazman, P. Kruchten, C. Verhoef, and H. van Vliet, editors, *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE Computer Society Press, 2001.

[67] M. A. Harrison and V. Maverick. Presentation by tree transformation. In *Proceeedings of 42nd IEEE International Computer Conference*, pages 68–73. IEEE Computer Society Press, 1997.

[68] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[69] G. T. Heineman and W. T. Council. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[70] I. Herman and M. S. Marshall. Graphxml - an XML-based graph description format. In J. Marks, editor, *Proceedings of the Symposium on Graph Drawing (GD 2000)*, volume 1984 of *Lecture Notes in Computer Science*, pages 52–62. Springer-Verlag, 2001.

[71] A. van der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf. Software release management. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 159–175. Springer-Verlag / ACM Press, 1997.

[72] International Telecommunications Union. *ITU-T Recommendation Z.100, Specification and Description Language (SDL)*, 1988.

[73] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[74] S. Jarzabek and R. Seviora. Engineering components for ease of customization and evolution. *IEE Proceedings – Software Engineering*, 147(6):237–248, December 2000. A special issue on Component-based Software Engineering.

[75] P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):1–34, 2000.

[76] S. C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[77] H. A. Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. Technical Report SEN-R0212, CWI, 2002.

[78] M. de Jonge. boxenv.sty: A LaTeX style file for formatting BOX expressions. Technical Report SEN-R9911, CWI, 1999.

[79] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, June 2000.

[80] M. de Jonge. Pretty-printing for software reengineering. In *Proceedings; IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 550–559. IEEE Computer Society Press, October 2002.

[81] M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, April 2002.

[82] M. de Jonge and Ramin Monajemi. Cost-effective maintenance tools for proprietary languages. In *Proceedings; IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 240–249. IEEE Computer Society Press, November 2001.

[83] M. de Jonge, E. Visser, and J. Visser. Collaborative software development. Technical Report SEN-R0113, CWI, 2001.

[84] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. G. J. van den Brand and D. Parigot, editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 211–218. Elsevier Science Publishers, April 2001.

[85] M. de Jonge and J. Visser. Grammars as contracts. In G. Butler and S. Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99, Erfurt, Germany, October 2001. Springer-Verlag.

[86] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[87] B. W. Kernighan. The Unix system and software reusability. *IEEE Trans. on Software Engineering*, SE-10(5):513–518, September 1984.

[88] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[89] D. E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Stanford University Center for the Study of Language and Information, 1992.

[90] J. Kort, R. Lämmel, and C. Verhoef. The grammar deployment kit. In M. G. J. van den Brand and Ralf Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65-3. Elsevier Science Publishers, 2002.

[91] J. Kort, R. Lämmel, and J. Visser. Functional Transformation Systems. In *9th International Workshop on Functional and Logic Programming*, Benicassim, Spain, July 2000.

[92] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[93] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. G. J. van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[94] R. Lämmel. Grammar Adaptation. In J. N. Oliveira and Pamela Zave, editors, *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer-Verlag, 2001.

[95] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

[96] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[97] R. Lämmel and J. Visser. Typed combinators for generic traversal. In *PADL 2002: Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer-Verlag, 2002.

[98] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[99] Y. Lin and S. P. Reiss. Configuration management in terms of modules. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, volume 1005 of *Lecture Notes in Computer Science*, pages 101–117. Springer-Verlag, 1995.

[100] D. Mackenzie and B. Elliston. Autoconf: Generating automatic configuration scripts, 1998. Available at `http://www.gnu.org/manual/autoconf/`.

[101] D. Mackenzie and T. Tromey. Automake, 2001. Available at `http://www.gnu.org/manual/automake/`.

[102] V. Matena and M. Hapner. *Enterprise javaBeans Specification*. Sun Microsystems, 1.1 edition, December 1999.

[103] D. McIlroy. Mass-produced software components. In P. Naur and B. Randell, editors, *Software Engineering*, NATO Science Committe report, pages 138–155, 1968.

[104] P. Miller. Recursive make considered harmful. *AUUGN*, 19(1):14–25, 1998. Available at `http://aegis.sourceforge.net/auug97.pdf`.

[105] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.

[106] E. Morcos-Chounet and A. Conchon. PPML: a general formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier Science Publishers, 1986.

[107] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering Methodology*, 5(3):262–292, 1996.

[108] J. Neighbors. An assessment of reuse technology after ten years. In *Proceedings of the Third International Conference on Software Reuse*, pages 6–13. IEEE Computer Society Press, 1994.

[109] Object Managament Group, Inc (OMG), 492 Old Connecticut Path, Framingham, MA 01701. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, 2000.

[110] R. van Ommering. Configuration management in component based product populations. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.

[111] R. van Ommering and J. Bosch. Widening the scope of software product lines – from variation to composition. In G. J. Chastek, editor, *Proceedings: Second Software Product Line Conference (SPLC2)*, volume 2379 of *Lecture Notes in Computer Science*, pages 328–347. Springer-Verlag, 2002.

[112] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[113] D. C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[114] H. Ossher, W. Harrison, and P. Tarr. Software engineering tools and environments. In *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)*, pages 261–278. ACM Press, 2000.

[115] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[116] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–38, March 1979.

[117] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, July 1995.

[118] J. S. Poulin. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Addison-Wesley, 1997.

[119] J. S. Poulin and J. M. Caruso. Determining the value of a corporate reuse program. In *Proceedings of the International Software Metrics Symposium*, pages 16–27. IEEE Computer Society Press, May 1993.

[120] V. Rajlich. Incremental redocumentation with hypertext. In *1st Euromicro Working Conference on Software Maintenance and Reengineering CSMR 97*, pages 68–73. IEEE Computer Society Press, 1997.

[121] V. Rajlich. Incremental redocumentation using the web. *IEEE Software*, 17(5):102–106, September/October 2000.

[122] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[123] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[124] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[125] M. Ruckert. Conservative Pretty-Printing. *SIGPLAN Notices*, 23(2):39–44, 1996.

[126] M. Schmitt. The development of a parser for SDL-2000. Technical Report A-00-10, Medical University of Lübeck, 2000.

[127] M. P. A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In H. Yang and L. White, editors, *Proceedings; IEEE International Conference on Software Maintenance (ICSM 1999)*, pages 245–255. IEEE Computer Society Press, 1999.

[128] M. P. A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society Press, March 2000.

[129] D. Sharp and W. Roll. Pattern usage in an avionics mission processing product line. In *Proceedings of the OOPSLA 2001 Workshop on Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems*, 2001. Available at `http://www.cs.wustl.edu/~mk1/RealTimePatterns/`.

[130] M. Silva and C. Werner. Packaging reusable components using patterns and hypermedia. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 146–155. IEEE Computer Society Press, 1996.

[131] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[132] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[133] R. Takahashi, Y. Muraoka, and Y. Nakamura. Building software quality classification trees: Approach, experimentation, evaluation. In *8th Internal Symposium on Software Reliability Engineering (ISSRE 97)*, pages 222–233. IEEE Computer Society Press, 1997.

[134] R. C. Tausworthe. Information models of software productivity: Limits on productivity growth. *Journal of System Software*, 19(2):185–201, 1992.

[135] D. B. Tucker and S. Krishnamurthi. Applying module system research to package management. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.

[136] M. Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society Press, 2001.

[137] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[138] E. Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 1999.

[139] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

[140] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[141] E. Visser. *Tiger in Stratego: An Experiment in Compilation by Transformation*. Institute of Information and Computing Sciences, Utrecht University, 2001. Technical Documentation. Available at `http://www.stratego-language.org/tiger/`.

[142] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

[143] E. Visser et al. The online survey of program transformation. Available at `http://www.program-transformation.org/`.

[144] J. Vlissides. Pluggable factory, part I. *C++ Report*, November/December 1998.

[145] J. Vlissides. Pluggable factory, part II. *C++ Report*, February 1999.

[146] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices*, 34(9):148–159, September 1999. Proceedings of the International Conference on Functional Programming (ICFP'99), Paris, France.

[147] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–28, Berkeley, CA, October 1997. USENIX Association.

[148] H. Westra. Configurable transformations for high-quality automatic program improvement – CobolX: a case study. Master's thesis, Department of Computer Science, Utrecht University, January 2002. Appeared as technical report INF/SCR-02-01.

[149] D. Whitgift. *Methods and Tools for Software Configuration Management*. John Wiley & Sons, 1991.

[150] D. S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 472–480. ACM Press, 1997.

[151] A. Zeller and G. Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, 1997.

Summary in Dutch / Samenvatting

Het Onderzoek

Softwarehergebruik is een manier om de praktijk van softwareontwikkeling te verbeteren, door bij de ontwikkeling van nieuwe systemen gebruik te maken van reeds bestaande en getestte onderdelen (componenten). Dit versnelt de ontwikkeling van nieuwe softwaresystemen, aangezien minder software nieuw hoeft te worden ontwikkeld. Tevens verhoogt het de kwaliteit van softwaresystemen omdat onderdelen van erkende kwaliteit kunnen worden gebruikt.

Al sinds de jaren zestig denkt men na over een 'ideale' softwarewereld, die softwareontwikkelaars verdeelt in twee groepen: componentontwikkelaars en componentgebruikers (systeemontwikkelaars). De eerste groep houdt zich bezig met de ontwikkeling van kwalitatief hoogstaande en algemeen bruikbare componenten, ook wel aangeduid als "development *for* reuse" (ontwikkeling ten dienste van hergebruik). De tweede groep houdt zich bezig met de ontwikkeling van component-gebaseerde (*component-based*) softwaresystemen. Daarbij wordt gebruik gemaakt van door de eerste groep geleverde componenten. Deze aktiviteit noemt men "development *with* reuse" (onwikkeling met hergebruik). Aanbod van en vraag naar softwarecomponenten vinden elkaar, idealiter, op een uitgebreide componentenmarkt.

Tot op heden wordt deze ideale softwarewereld slechts op kleine schaal, in enkele specifieke toepassingsgebieden (domeinen), benaderd. Van een algemene componentenmarkt is dan ook nog lang geen sprake, want hoewel veelbelovend, blijkt succesvol hergebruik van softwarecomponenten erg moeilijk in de praktijk te brengen.

Om softwarehergebruik succesvoller te maken zijn specifieke technieken nodig die componentontwikkelaars helpen bij het bouwen van herbruikbare componenten en componentgebruikers bij het zoeken, selecteren en integreren ervan. Deze technieken voor softwarehergebruik dienen te voldoen aan vier criteria, ze moeten:

1. een voldoende mate van abstractie bieden voor herbruikbare onderdelen;

2. hergebruik van onderdelen eenvoudiger maken dan het opnieuw ontwik-
   kelen ervan;

3. duidelijkheid verschaffen over de functionaliteit van herbruikbare onder-
   delen;

4. het vinden van herbruikbare onderdelen sneller en eenvoudiger maken
   dan het opnieuw ontwikkelen ervan.

Dit proefschrift beschrijft onderzoek dat ten doel heeft om effectief software-
hergebruik mogelijk maken. Daartoe worden hergebruiktechnieken ontwik-
keld die aan de bovengenoemde criteria voldoen. De drie centrale thema's die
hierbij een rol spelen zijn: abstractie, compositie en granulariteit.

Abstractie Diverse studies tonen aan dat softwarehergebruik in de praktijk
moeilijk is. De reden is een gebrek aan algemene abstracties, anders dan lijs-
ten (*lists*), stapels (*stacks*), wachtrijen (*queues*) etc. Wanneer men zich echter
richt op een specifiek toepassingsdomein, dan zijn (domein-specifieke) abstrac-
ties vaak wel duidelijk te onderscheiden. Binnen zo'n domein zou software-
hergebruik derhalve succesvol kunnen zijn. In dit proefschrift onderzoeken
we deze hypothese en proberen we binnen het domein van taalverwerking
(*language processing*) een effectieve hergebruikpraktijk te ontwikkelen. Typi-
sche abstracties die in dit domein een rol spelen zijn: ontleden (*parsing*), ver-
talen (*compiling*), transformeren (*transforming*) en opmaken (*pretty-printing*).

Compositie De ontwikkeling van softwaresystemen door middel van her-
bruikbare componenten impliceert softwarecompositie. Immers, vanwege her-
gebruik kan een softwaresysteem niet als één enkel geheel beschouwd wor-
den. In plaats daarvan vormt het een compositie van enerzijds hergebruikte
en anderzijds specifiek-geschreven onderdelen. Componenten kunnen in ver-
schillende programmeertalen geschreven zijn. Componenten kunnen daar-
naast verschillende compositiemomenten hebben. Deze compositiemomen-
ten leiden tot een onderscheid in verschillende typen componenten, bijvoor-
beeld: broncodecomponenten die tijdens compilatie samengevoegd worden en
binaire componenten die pas tijdens de executie van een applicatie samen-
gevoegd worden tot één geheel. Het is wenselijk dat applicaties eenvoudig
gebruik kunnen maken van componenten die in verschillende programmeer-
talen zijn geschreven én dat verschillende compositiemomenten gecombineerd
kunnen worden. Daartoe ontwikkelen we in dit proefschrift een architectuur
die verschillende integratiemomenten van componenten, geschreven in diverse
programmeertalen, ondersteunt. Tevens onderzoeken we hoe de compositio-
naliteit van componenten verhoogd kan worden door instituut-specifieke af-
hankelijkheden te vermijden en door het compositieproces te automatiseren.

Granulariteit Hoe groot moet een component zijn, hoeveel functionaliteit moet het bevatten en hoe specifiek moet die functionaliteit zijn? Deze vragen met betrekking tot componentgranulariteit zijn niet eenduidig te beantwoorden. Granulariteit bepaalt echter in grote mate de herbruikbaarheid van componenten. Zo neemt grofkorrelig hergebruik de componentgebruiker veel werk uit handen omdat componenten relatief groot zijn en veel specialistische functionaliteit bevatten. Deze vorm van hergebruik kan dan ook eenvoudig tot kostenbesparingen leiden. Helaas is het aantal toepassingen van zulke specialistische componenten beperkt. Ook is de kans op codeduplicatie groot omdat potentieel herbruikbare onderdelen vast verweven kunnen zitten in een component en derhalve niet herbruikbaar zijn.

Tegenover grofkorrelig hergebruik staat fijnkorrelig hergebruik. Hierbij zijn componenten relatief klein en hebben beperkte, algemeen bruikbare functionaliteit. Dit verhoogd de herbruikbaarheid van componenten en het kan codeduplicatie helpen te verminderen. De voordelen van fijnkorrelig hergebruik zijn voor een componentgebruiker echter geringer vanwege de beperkte functionaliteit die per component hergebruikt wordt en vanwege het algemene karakter van die functionaliteit. Daarnaast groeit de complexiteit van het softwareontwikkelproces naarmate het aantal gebruikte componenten toeneemt.

De titel van dit proefschrift, "To Reuse or To Be Reused", heeft betrekking op deze tegenstrijdigheid: grofkorrelig hergebruik, dat specifieke voordelen biedt voor componentgebruikers en fijnkorrelig hergebruik, dat juist aan de componentontwikkelaar voordelen biedt. In dit proefschrift onderzoeken we of deze tegenstrijdigheid opgeheven kan worden om zo de voordelen van grof- en fijnkorrelig hergebruik te combineren.

De Hoofdstukken

Dit proefschrift bestaat uit twee delen. Het eerste deel ("onwikkeling ten dienste van hergebruik") richt zich op de componentontwikkelaar. Het behandelt onderzoek ter verbetering van de herbruikbaarheid van softwareonderdelen (componenten). Het domein van taalverwerking is hierbij gekozen als specifiek toepassingsdomein omdat het, zo is de aanname, voldoende domeinabstracties biedt voor succesvol hergebruik. In hoofdstuk 2, "Grammars as Contracts", ontwikkelen we een raamwerk voor componenthergebruik waarbij grammatica's een centrale rol vervullen. Enerzijds definiëren zij de gegevensstructuren van informatie die componenten onderling uitwisselen. Anderzijds dienen zij om taalafhankelijke onderdelen te genereren om zodoende onderhoud aan componenten, als gevolg van taalwijzigingen, te minimaliseren. Het hoofdstuk presenteert tevens een softwareontwikkelmodel voor de ontwikkeling van component-gebaseerde taalgereedschappen. Dit ontwikkelmodel heet "Language-Centered Software Engineering" (taal-georiënteerde softwareontwikkeling), of kortweg, LCSE. Hoofdstuk 3 beschrijft de componentenbundel XT die de implementatie van het raamwerk vormt. XT is een verzameling com-

ponenten gericht op de ontwikkeling van programmatransformatiesystemen volgens het ontwikkelmodel van LCSE. Naast XT zelf, beschrijft het hoofdstuk ook een methode om hergebruik van XT componenten te meten. Deze meet-methode wordt gedurende het hele proefschrift gebruikt om de effectiviteit van de diverse hergebruiktechnieken te achterhalen. Hoofdstuk 4, "Pretty-Printing for Software Reengineering", beschrijft onderzoek op het gebied van pretty-printing, d.w.z. het opmaken van programmateksten. Pretty-printing vormt een integraal onderdeel van LCSE. Dit stelt harde eisen aan pretty-print componenten wat betreft herbruikbaarheid. Het hoofdstuk onderzoekt deze specifieke eisen en presenteert de benodigde technieken om aan deze eisen te voldoen.

Deel twee van dit proefschrift ("ontwikkeling met hergebruik") richt zich op de componentgebruiker. Onderzocht wordt of de in deel één ontwikkelde technologieën succesvol zijn en hoe assemblage en configuratie van samengestelde softwaresystemen geautomatiseerd kunnen worden. In hoofdstuk 5, "Cost-Effective Maintenance Tools for Proprietary Languages", worden de ontwikkelde technieken uit deel één in de praktijk toegepast in een industrieel samenwerkingsverband voor de ontwikkeling van een documentatiegenerator. Onderzocht wordt of, zoals LCSE belooft, softwarehergebruik- en generatie de ontwikkeling van taal-georiënteerde softwaresystemen kan bespoedigen en vereenvoudigen. Hoofdstuk 6, "Source Tree Composition", stelt dat naast functionele compositie, ook compositie van broncodemodulen, softwarebouw- en configuratiepocessen nodig is voor de ontwikkeling van component-gebaseerde softwaresystemen. Daartoe wordt de techniek "Source Tree Composition" gepresenteerd die deze vorm van compositie definieert en automatiseert. Source tree compositie heeft ten doel om automatische assemblage van softwaresystemen binnen handbereik te brengen. Hierbij is selectie van gewenste systeemonderdelen voldoende om automatisch de bijbehorende implementatieonderdelen te vinden en te integreren. Hoofdstuk 7, "Feature-Based Product Line Instantiation using Source-Level Packages", diept het onderwerp van automatische softwareassemblage verder uit. Dat hoofdstuk onderzoekt algemene technieken voor het opzetten en implementeren van software produktlijnen. Deze technieken behelzen het automatisch vertalen van klant-specifieke, marktgerelateerde produkteigenschappen (*features*) naar technische aspecten die de corresponderende implementatie vormen. De omzetting van de commerciële documentatiegenerator DOCGEN naar een produktlijnarchitectuur vormt het uitganspunt van dit onderzoek.

De Resultaten

In hoofdstuk 8 worden de resultaten van dit proefschrift beschreven aan de hand van de onderzoeksvragen die betrekking hebben op de drie centrale thema's van dit proefschrift: abstractie, compositie en granulariteit. Hieronder volgt een korte samenvatting van de behaalde resulaten. Tabel 8.1 op

bladzijde 140 geeft een overzicht van de ontwikkelde hergebruiktechnieken, de hoofdstukken waar ze worden gepresenteerd, alsmede de hergebruikcriteria waaraan ze voldoen.

Abstractie   Effectief softwarehergebruik in het domein van taalverwerking is mogelijk. Hoofdstuk 5, "Cost-Effective Maintenance Tools for Proprietary Languages", demonstreert dat softwarehergebruik met behulp van het in hoofdstuk 2, "Grammars as Contracts", beschreven ontwikkelmodel en de in hoofdstuk 3, "XT: a Bundle of Program Transformation Tools", beschreven componenten, efficiënte softwareonwikkeling in het domein van taalverwerking mogelijk maakt.

Compositie   Voor effectief softwarehergebruik zijn geschikte compositietechnieken onontbeerlijk. Het onderzoek dat ten grondslag ligt aan dit proefschrift heeft geresulteerd in compositietechnieken op drie verschillende vlakken. In hoofdstuk 2, "Grammars as Contracts", beschrijven we compositietechnieken op het functionele vlak binnen het taalverwerkingsdomein. De meest opvallende techniek is het gebruik van grammatica's als kontrakt tussen componenten. Source tree compositie, gepresenteerd in hoofdstuk 6, is een compositietechniek op het vlak van broncodecomponenten, gericht op automatische softwareassemblage. Tenslotte worden in hoofdstuk 7, "Feature-Based Product Line Instantiation using Source-Level Packages", technieken gepresenteerd voor de compositie van produkteigenschappen. Deze vorm van compositie behelst het samenstellen van markt-gerelateerde produkteigenschappen (features) tot operationele softwareprodukten.

Granulariteit   Door de korrelgrootte van softwarecomponenten variabel te maken kunnen de voordelen van fijn- en grofkorrelig hergebruik gecombineerd worden. De in hoofdstuk 6, "Source Tree Composition", gepresenteerde techniek ondersteunt zo'n variabele korrelgrootte voor broncodecomponenten. Enerzijds staat deze techniek componentontwikkelaars toe om kleine, algemeen bruikbare, componenten te ontwikkelen. Anderzijds staat het componentgebruikers toe om grote componenten her te gebruiken die intern echter een fijnkorrelige structuur kunnen hebben.

De effectiviteit van de in dit proefschrift beschreven technieken voor softwarehergebruik wordt aangetoond aan de hand van hergebruikpercentages die in de diverse hoofdstukken zijn berekend. Tabel 8.1 op bladzijde 146 bevat een overzicht van deze meetresultaten. Het gemeten hergebruikpercentage tussen 80 en 91 procent is hoog in vergelijking met andere studies en toont aan dat de technieken die in dit proefschrift beschreven zijn een nuttige aanvulling vormen op bestaande hergebruiktechnieken.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02