To reuse or to be reused. Techniques for component composition and construction

de Jonge, M.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
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 it 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 than “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].

**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

---

1Observe 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-print(goto-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.
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.
Introduction

Chapter 1

<table>
<thead>
<tr>
<th>Quality</th>
<th>Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Coarse-grained</td>
</tr>
<tr>
<td>Payoff</td>
<td>+</td>
</tr>
<tr>
<td>Cognitive distance</td>
<td>+</td>
</tr>
<tr>
<td>Software construction process</td>
<td>+</td>
</tr>
<tr>
<td>General usability</td>
<td>−</td>
</tr>
<tr>
<td>Code duplication across components</td>
<td>−</td>
</tr>
</tbody>
</table>

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].