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
Conclusions

<table>
<thead>
<tr>
<th>Technique (chapter)</th>
<th>Truism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar Base (2)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Library Generation (2)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Program Generation (2)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Generic Tools (2)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Grammars as contracts (2)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Standardized exchange formats (2)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Abstract from concrete syntax (2)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>XT bundle (3)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Generic pretty-printing (4)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Source tree composition (6)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Build interfaces (6)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Configuration interfaces (6)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Package definitions (6)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Online package base (6)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Feature definition language (7)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Customer factories (7)</td>
<td>✓✓✓✓</td>
</tr>
<tr>
<td>Online feature base (7)</td>
<td>✓✓✓✓</td>
</tr>
</tbody>
</table>

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).

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 com-
ponents 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 soft-
ware development easier because the software construction process becomes
uniform, and because build processes as well as configuration processes of dif-
ferent 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 con-
tribution form at an online package base.

As part of our research, we initiated the Online Package Base (see Ap-
pendix A), which forms a central meeting point for developers and users of
source packages. It provides online package selection, bundling, and contribu-
tion 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 differ-
ent 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 com-
bined?

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 composi-
tionality 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 de-
pendencies upon other source code components.
With build interfaces, configuration interfaces, and package definitions, coarse-grained components can easily be split 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.
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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Non-transitive reuse LOC</th>
<th>RSI</th>
<th>Reuse%</th>
<th>Transitive reuse LOC</th>
<th>Reuse%</th>
</tr>
</thead>
<tbody>
<tr>
<td>aterm-tools</td>
<td>3,616</td>
<td>2,999</td>
<td>82%</td>
<td>3,616</td>
<td>82%</td>
</tr>
<tr>
<td>autobundle</td>
<td>6,386</td>
<td>4,687</td>
<td>73%</td>
<td>10,273</td>
<td>83%</td>
</tr>
<tr>
<td>gpp</td>
<td>12,178</td>
<td>8,408</td>
<td>69%</td>
<td>26,349</td>
<td>85%</td>
</tr>
<tr>
<td>grammar-recovery</td>
<td>7,871</td>
<td>6,382</td>
<td>81%</td>
<td>7,871</td>
<td>81%</td>
</tr>
<tr>
<td>graph-tools</td>
<td>3,002</td>
<td>2,646</td>
<td>88%</td>
<td>3,002</td>
<td>88%</td>
</tr>
<tr>
<td>sdf-tools</td>
<td>23,700</td>
<td>20,601</td>
<td>86%</td>
<td>61,186</td>
<td>94%</td>
</tr>
<tr>
<td>stratego-tools</td>
<td>8,667</td>
<td>7,091</td>
<td>81%</td>
<td>8,667</td>
<td>81%</td>
</tr>
<tr>
<td>xt</td>
<td>243</td>
<td>0</td>
<td>0%</td>
<td>29,445</td>
<td>99%</td>
</tr>
<tr>
<td>asfix-tools</td>
<td>6,442</td>
<td>4,433</td>
<td>68%</td>
<td>6,442</td>
<td>68%</td>
</tr>
<tr>
<td>sdl-tools</td>
<td>5,043</td>
<td>4,487</td>
<td>88%</td>
<td>22,819</td>
<td>97%</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td><strong>77,148</strong></td>
<td><strong>61,734</strong></td>
<td><strong>80%</strong></td>
<td><strong>179,670</strong></td>
<td><strong>91%</strong></td>
</tr>
</tbody>
</table>