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. 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 sub-scenarios.

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 inter-operation 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 SDG 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

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 aterm 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 reuse 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 incoming 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 a 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:

\[
\text{reuse level} = \frac{\text{Reused Software}}{\text{Total Software}} \times 100\%
\]

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:

\[
\text{Reuse\%} = \frac{\text{RSI}}{\text{Total Statements}} \times 100\%
\]
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. 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.

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 than 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-
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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Non-transitive reuse</th>
<th>Transitive reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>RSI</td>
</tr>
<tr>
<td>atermdiff</td>
<td>1,221</td>
<td>1,065</td>
</tr>
<tr>
<td>yacc2sdf</td>
<td>2,427</td>
<td>1,905</td>
</tr>
<tr>
<td>GraphXML2dot</td>
<td>1,054</td>
<td>863</td>
</tr>
<tr>
<td>sdf2asdf</td>
<td>1,294</td>
<td>929</td>
</tr>
<tr>
<td>sdf-label</td>
<td>1,652</td>
<td>1,500</td>
</tr>
<tr>
<td>sdf2sg</td>
<td>1,606</td>
<td>1,286</td>
</tr>
<tr>
<td>tohtml-sdf</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>sdf-doc</td>
<td>1,001</td>
<td>897</td>
</tr>
<tr>
<td>pack-sdf</td>
<td>1,504</td>
<td>1,352</td>
</tr>
<tr>
<td>sdf-bracket</td>
<td>868</td>
<td>775</td>
</tr>
<tr>
<td>sdf2text</td>
<td>47</td>
<td>0</td>
</tr>
<tr>
<td>sdf2sdf</td>
<td>672</td>
<td>567</td>
</tr>
<tr>
<td>sdf-wf</td>
<td>1,350</td>
<td>1,089</td>
</tr>
<tr>
<td>sdf-imports</td>
<td>1,410</td>
<td>1,107</td>
</tr>
<tr>
<td>sdf2strategy</td>
<td>2,300</td>
<td>1,319</td>
</tr>
<tr>
<td>implode-asfix</td>
<td>1,708</td>
<td>693</td>
</tr>
<tr>
<td>pp</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>parse</td>
<td>53</td>
<td>0</td>
</tr>
<tr>
<td>gbquery</td>
<td>112</td>
<td>0</td>
</tr>
</tbody>
</table>

**Totals:** 20,362 15,347 75% 68,776 92%

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 be 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 OPEN C++ [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.

**Acknowledgments** XT bundles the efforts of several people: ATERM library (P. Olivier, H. de Jong), GPP (M. de Jonge), SDF2 (E. Visser), sgdr (E. Visser, J. Scheerder, M. van den Brand), pgen (E. Visser, M. van den Brand), Stratego (E. Visser), Grammar Tools (M. de Jonge, E. Visser and J. Visser). The Grammar Base was initiated by M. de Jonge, E. Visser and J. Visser and incorporates grammars constructed at UvA, CWI, and UU over a period of several years.

We thank Paul Klint, Tobias Kuipers, and Jurgen Vinju for their helpful comments on a draft of the chapter.