XT: a bundle of program transformation tools

Merijn de Jonge & Eelco Visser & Joost Visser

CWI, Department SEN
PO Box 94079, 1090 GB Amsterdam, The Netherlands
Merijn.de.Jonge@cwi.nl
Joost.Visser@cwi.nl

Universiteit Utrecht
Institute of Information and Computing Sciences
PO Box 80.089, 3508 TB Utrecht, The Netherlands
visser@acm.org

XT bundles existing and newly developed program transformation libraries and tools into an open framework that supports component-based development of 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.

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. It bundles its constituents into an *open framework* for *component-based* transformation tool development, which is flexible and extendible. XT is free software.

In this short paper, we will provide a brief overview of XT and an indication of what is possible with it. Section 2 fixes terminology and discusses various common program transformation *scenarios*. Section 3 outlines the program transformation development process that we want to support. Section 4 discusses the actual content of the XT bundle, and explains how the various constituents of XT can be used to support various program transformation development tasks. Section 5 summarizes our experiences with XT so far, and Section 6 wraps up with concluding remarks.

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* and *target* languages respectively. Below we will distinguish scenarios where the source 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* that are characterized by the *constraints* imposed on the main scenarios.

Translation In a *translating* scenario a program is transformed from a source language into a program in a *different* target language. Examples of translating scenarios are synthesis, migration, compilation, analysis. In program *synthesis* an implementation is derived from a high-level specification by semantics preserving transformations. That is, it is guaranteed that the implementation satisfies the specification. A prime example of program synthesis is parser generation. In *migration* a program is transformed to another language, while preserving all meaning. 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 more low-level language. In program *analysis* a program is reduced to some property, or value. 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. 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* some aspect of a program is improved. For example, repairing a Y2K bug. A *refactoring* is a meaning preserving modification that improves the design. An *optimization* is a meaning preserving modification that improves the run-time and/or space performance of the program.

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. Transformation development

The development process of program transformation tools generally takes places along the following steps:

1. Obtain (syntax) definitions of the languages involved in the transformation. This may involve (re)construction of grammars, transformation of grammars, assessment of existing grammars, in short: grammar engineering.
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. Perform the transformations.

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.

4. The XT bundle

XT bundles tooling for the construction of program transformation systems. Its purpose is to minimize installation effort, 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 [3] — 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 two representations for ATERMS: a human-readable, textual representation and a space efficient binary representation based on maximal subtree sharing. Furthermore, a library of functions for building, traversing, and inspecting ATERMS is available.

- SDF [9, 12] — All grammars bundled with XT are defined in the modular syntax definition formalism SDF. Parsing of languages defined in SDF is supported by the parser generator `pgen` and the generic parser `sglr`. The parser generator produces parse tables that are interpreted by `sglr` using the Scannerless Generalized-LR parsing algorithm.

- GPP [4] — Pretty-printing is supported by the generic pretty-print toolset GPP. 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, but the system can be extended easily to support more output formats.

- Grammar Base [7] — The SDF Grammar Base contains a collection of syntax definitions for a growing number of languages, including COBOL, HASKELL, YACC, SDF, and ELAN. The purpose of the Grammar Base is to offer a reference for language definitions and to provide a collection of grammars that can be downloaded for free and are ready for use.

- Grammar Tools [6] — We developed a collection of grammar-related tools including tools for grammar analysis, grammar (re)construction, and tree manipulation. The tools are themselves examples of *translating* and *rephrasing* program transformations.

- Stratego [14] — Stratego is a programming language for strategic term rewriting. The language 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.

- JJForester [11] — JJForester is a parser and visitor generator for Java which takes SDF as input, and uses `pgen` and `sglr` as front-ends. It allows implementation of specific program transformations by refinement of generated visitors.

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. The XT documentation 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 can be extended with new components and that existing components can be replaced at any time.

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 [1] was developed as an exercise in compilation by transformation for a course on *High-Performance Compilers* at Universiteit Utrecht [13]. 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 typechecker 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 [10]. The transformation system consists of a parser, a normalization phase to eliminate syntactic sugar, a typechecker, the warm fusion transformation itself and a pretty-printer. 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 grammar for HASKELL98 has been 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 makes 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 [5]. 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 visualisation 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.

6. Concluding remarks

Availability XT and all its constituent components are *free software* [16], i.e. they are distributed as open source under the GNU General Public Licence [8], and anyone is allowed to use, modify, and redistribute them. XT can be downloaded from http://www.program-transformation.org/xt. The distribution makes use of *autobundle*, *autoconf*, and *automake*, which make installation a nearly trivial job. XT is known to install and run successfully on various platforms, among which SUN-Solaris, BSD-Unix, and Linux.

Comparison to other frameworks XT shares its bundling infrastructure and a few of its constituent packages with a peer bundle: the ASF+SDF Meta-environment [2], an *integrated, interactive environment* for language prototyping. By contrast, XT bundles a partially overlapping set of packages into an *open framework* for *component-based* transformation tool development.

Many tools and frameworks for program transformation, or for some of its (sub-)scenarios, already exist. Among these are attribute grammar systems, algebraic rewriting systems, and object-oriented systems (see [15] for an overview of transformation frameworks). Generally, these systems are closed in the sense that they provide a fixed set of tightly-coupled components (parser, pretty-printer, 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, that 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.

Acknowledgments XT bundles the efforts of several people: ATERM library (P. Olivier, H. de Jong), GPP (M. de Jonge, M. van den Brand, E. Visser), SDF2 (E. Visser), `sglr` (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) JJForester (T. Kuipers and J.Visser). The Grammar Base was initiated by M. de Jonge, E. Visser and J. Visser and incorporates grammars constructed at UvA and CWI over a period of several years.

Merijn de Jonge and Joost Visser received partial support from the Telematica Institute, under the *Domain-specific Languages* project.

Bibliography

[1] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] M.G.J. van den Brand et al. The ASF+SDF Meta-Environment: a component-based language development environment. Submitted for publication.

[3] 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.

[4] M. de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*. University of Wollongong, Australia, 2000.

[5] M. de Jonge and R. Monajemi. Grammar re-engineering for language centered software engineering. Submitted for publication, oct 2000.

[6] M. de Jonge, E. Visser, and J. Visser. Grammar Tools. http://www.program-transformation.org/gt/.

[7] M. de Jonge, E. Visser, and J. Visser. The Grammar Base. http://www.program-transformation.org/gb/.

[8] GNU General Public License. http://www.fsf.org/copyleft/gpl.html.

[9] 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.

[10] 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*. (To appear).

[11] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. Submitted for publication.

[12] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[13] E. Visser. Tiger in Stratego: An exercise in compilation by transformation. http://www.stratego-language.org/tiger/, 2000.

[14] E. Visser, Z. 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).

[15] E. Visser et al. The online survey of program transformation. http://www.program-transformation.org/survey.html.

[16] What is Free Software? http://www.fsf.org/philosophy/free-sw.html.