Stratego/XT 0.17. A Language and Toolset for Program Transformation

M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser

TUDelft

SE**RG**

TUD-SERG-2008-011

This paper is a pre-print of:

M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*. Elsevier, 2008.

```
@article{BKVV08,
  author   = {Martin Bravenboer and Karl Trygve Kalleberg
               and Rob Vermaas and Eelco Visser},
  title    = {{Stratego/XT 0.17}. {A} Language and Toolset for
               Program Transformation},
  journal  = {Science of Computer Programming},
  year     = {2008},
  publisher = {Elsevier},
  note     = {Special issue on Experimental Systems and Tools}
}
```

Stratego/XT 0.17. A Language and Toolset for Program Transformation

Martin Bravenboer [a,1] Karl Trygve Kalleberg [b,2] Rob Vermaas [c]
Eelco Visser [a,*]

[a] *Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science (EWI), Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands*

[b] *Department of Informatics, University of Bergen, PB 7800 N-5020 Bergen, Norway*

[c] *Machina, Utrecht, The Netherlands*

Stratego/XT is a language and toolset for program transformation. The Stratego language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high level of abstraction. The XT toolset offers a collection of flexible, reusable transformation components, and tools for generating such components from declarative specifications. Complete program transformation systems are composed from these components.

This paper gives an overview of Stratego/XT 0.17, including a description of the Stratego language and XT transformation tools; a discussion of the implementation techniques and software engineering process; and a description of applications built with Stratego/XT.

* Corresponding author.
  Email addresses: `martin.bravenboer@gmail.com` (Martin Bravenboer), `karltk@strategoxt.org` (Karl Trygve Kalleberg), `rob.vermaas@gmail.com` (Rob Vermaas), `visser@acm.org` (Eelco Visser).

1 Introduction

Automatic program transformation and *generative programming* aim at increasing programmer productivity by automating programming tasks using some form of automatic program generation or transformation, such as code generation from a domain-specific language, aspect weaving, optimization, or specialization of a generic program to a particular context. Key for achieving this aim is the construction of *tools* that implement the automating transformations. If generative programming is to become a staple ingredient of the software engineering process, the construction of generative tools itself should be automated as much as possible. This requires an infrastructure with support for the common tasks in the construction of transformation systems.

Stratego/XT is a generic infrastructure for creating stand-alone transformation systems [47,52]. It combines *Stratego*, a language for implementing transformations based on the paradigm of programmable rewriting strategies, with *XT*, a collection of reusable components and tools for the development of transformation systems. In general, Stratego/XT is intended for the *analysis*, *manipulation* and *generation* of programs, though its features make it useful for transforming any structured documents.

Reusability at all levels of granularity has been a leading principle in the design and implementation of Stratego/XT [47]. First, the focus on *transformation components* strongly promotes reuse of large-grained components. In many cases, users of Stratego/XT do not start with the development of a parser, but can immediately get started with the actual transformation. Stratego/XT has a varied selection of actively developed front-ends (Section 5). Second, the use domain-specific languages for different phases of a transformation system is a substantial time saver. In addition to the Stratego language itself, which is a DSL for transformation proper, Stratego/XT provides and/or integrates DSLs for aspects such as syntax definition, pretty-printing, term schemas, and unit testing. In this way, implementations are more abstract, easier to maintain, and easier to read. Third, the extensive Stratego library, with its generic traversals, generic transformations for scoping, control- and data-flow and many other convenience functions for program transformation allows developers to write their transformations concisely.

The component layers in Figure 1 illustrate the organization of Stratego/XT from the point of view of reusability, featuring five levels of components ranging from completely generic via language specific to application specific.

(1) At the bottom layer is the *substrate* for a transformation system, that is the data representation and exchange format, for which we use the Annotated Term Format (ATerm) [36] as basis, and XML where necessary to
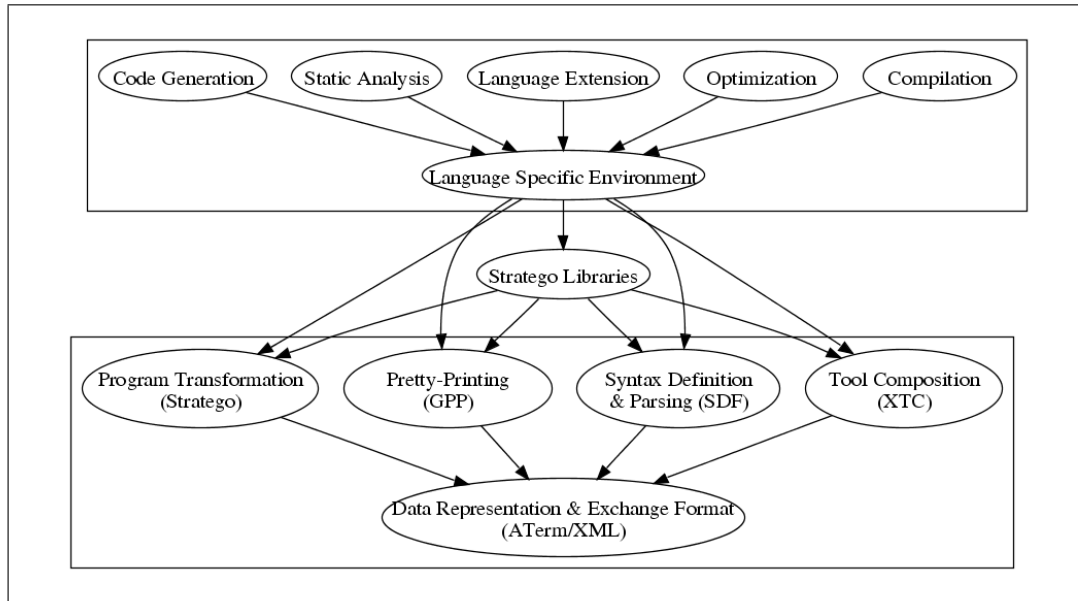
Fig. 1. Reuse layers in Stratego/XT.

communicate with external tools.

(2) The *foundation* of any transformation system are syntax definition and parsing, pretty-printing, program transformation, and tool composition. The syntax definition formalism SDF provides modular syntax definition and parsing, supporting easy combination of languages. The pretty-printing package GPP supports rendering structured program representations as program text. The program transformation language Stratego supports concise implementation of program transformations by means of rewrite rules and programmable strategies for control of their application. Finally, the XTC library supports composition of transformations implemented as independently executable tools. These are all generic facilities that are needed in any transformation for any language.

(3) In the middle is a library of transformations and transformation utilities that are not specific for a language, but not usable in all transformations either. The Stratego Libraries provide a host of generic rewriting strategies and utilities for generating parts of a transformation system.

(4) Near the top are specializations of the generic infrastructure to specific object languages. Such a *language specific environment* consists of a syntax definition for a language along with utilities such as semantic analysis, variable renaming, and module flattening.

(5) Finally, at the top are the actual *transformation systems* such as compilers, language extensions, static analysis tools, and aspect weavers. These tools are implemented as compositions of tools from the lower layers extended with components implementing the specific transformation under consideration.

Stratego/XT has been used to build many types of transformation systems including compilers, interpreters, static analyzers, domain-specific optimizers, code generators, source code refactorers, documentation generators, and document transformers. These systems involved numerous types of transformations, including desugaring of syntactic abstractions; assimilation of language embeddings [17]; bound variable renaming; optimizations, such as function inlining; data-flow transformations such as constant propagation, copy propagation, common-subexpression elimination, and partial evaluation [14,34]; instruction selection [16]; and several analyses including type checking [15] and escaping variables analysis.

This paper gives an overview of the design, implementation, and use of Stratego/XT 0.17. Section 2 outlines the technical foundations of the Stratego language and describes the compiler and interpreter that implement it. Section 3 describes the transformation infrastructure provided by the XT tool set. Section 4 examines the implementation techniques and methods used in the construction of Stratego/XT, which is used to implement itself. Furthermore, the tool supported software engineering process, including automatic release management and user support, is described. Section 5 outlines the experience with using Stratego/XT in concrete projects. Section 6 discusses previous and related work. It should be noted that this paper provides an overview of a large system. A detailed and complete techical discussion is beyond the scope of this paper. Where applicable, we refer to previous work for more details.

2   The Stratego Language

In transformation systems built with Stratego/XT, transformation components are implemented using the Stratego language [51,47,14]. Stratego provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations.

2.1   Terms

Stratego programs transform *first-order terms*. In particular, Stratego programs transform ATerms [36], a term format designed for exchange of structured data between transformation programs, which is supported by libraries for internal (in memory) representation of terms. Essentially, Stratego terms are isomorphic with structures according to the following definition:

$$t := c(t_1, ..., t_n)$$

that is, a term is an application of a constructor c to zero or more terms t_i. In practice, the syntax is a bit richer, i.e., terms (or rather pre-terms) are defined as

$$pt := s \mid i \mid f \mid c(t_1, ..., t_n) \mid [t_1, ..., t_n] \mid (t_1, ..., t_n)$$

including special notation for string (s), integer (i), and float (f) constants, and for lists ([]), and tuples (()). Furthermore, terms can be extended with a list of *annotations*, which are themselves terms.

$$t := pt \mid pt\{t_1, ..., t_n\}$$

Annotations can be used to add additional (semantic) information to a term.

Terms are equivalent to trees, i.e. directed acyclic graphs with ordered outgoing edges, and are used to represent parse or abstract syntax trees of programs, or any other structured documents. For example, `Call("square", [Plus(Var("x"), Int(3))])` is a typical term representation of an expression `square(x + 3)`. Stratego is agnostic to the producers and consumers of terms, i.e. Stratego is not concerned with turning programs into terms, or vice versa. In next section we discuss how the XT tools support the creation of parsers and pretty-printers, which can be connected to Stratego transformations.

Stratego requires the declaration of term constructors used in a program by means of a signature. A signature defines for each constructor the result type (sort) and the types (sorts) of its arguments. For example, the following signature declares a couple of constructors typical in an expression language:

```
signature
  sorts Id Exp
  constructors
    Var  : Id -> Exp
    Plus : Exp * Exp -> Exp
    If   : Exp * Exp * Exp -> Exp
```

The current version of Stratego only checks that constructors are declared and have the right arity. Argument types are not checked statically. Stratego also supports *overlays*, a mechanism for declaring *pseudo-constructors* [43]. That is, constructors defined in terms of other constructors. For example, the definition of a short-circuit disjunction operator `Or`, can be defined in terms of the `If` constructor by means of the following overlay definition:

```
overlays
  Or(e1, e2) = If(e1,True(),e2)
```

This is essentially a macro definition for constructors.

2.2 Term Rewrite Rules

Rewrite rules are the basic units of transformation. A rewrite rule has the form $R : p_1 \rightarrow p_2$, where R is the name of the rule, p_1 the *left-hand side pattern* of the rule and p_2 the *right-hand side pattern*. A pattern is a term with variables. For example, the rule

```
PlusZero : Plus(e, Int(0)) -> e
```

defines that `0` is a right unit for addition. Here `e` is a term *variable*.

Applying a rule R to a term t entails matching p_1 against t, binding the variables in the pattern. If they match, the rule replaces t with the instantiation of the right-hand side p_2, replacing its variables with the terms found during matching. Pattern matching in Stratego is basic first-order matching (one way unification).

Conditional rewrite rules impose a further constraint on the applicability of a rule. A conditional rule $R : p_1 \rightarrow p_2$ `where` s, applies to a term if the left-hand side p_1 matches *and* the condition s succeeds. An example of a conditional rule is the following constant folding rule

```
EvalBinOp : Plus(Int(i), Int(j)) -> Int(k) where k := <add>(i,j)
```

which evaluates the arguments of an addition operator, replacing the additive expression with the result of the evaluation.

2.3 Concrete Object Syntax

While terms are a fine representation for programs, term patterns for realistic program fragments may become unwieldy. Therefore, Stratego supports the use of *concrete syntax* [46] in the patterns of rewrite rules. That is, rather than expressing abstract syntax tree patterns using nested constructor applications, one can use the concrete syntax of the object language. For example, the rewrite rules we saw above can be written as follows using concrete syntax:

```
PlusZero  : |[ e + 0 ]| -> |[ e ]|
EvalBinOp : |[ i + j ]| -> |[ k ]| where k := <add>(i, j)
```

To realize concrete syntax embedding, the syntax of Stratego is extended with the syntax of the object language. This embedding is completely configurable, including the notation for quotation, and can be declared to the compiler as a plugin using the declarative syntax definition formalism SDF. The embedding may also include the declaration of meta-variable schemes, such as used in the example above, where `e` is a meta-variable for expressions, and `i`, `j`, and `k` are meta-variables for integer constants. The compiler translates rules using concrete syntax to regular rules using abstract syntax terms.

2.4 Programmable Rewriting Strategies

Traditional *term rewriting* is the exhaustive application of a set of rewrite rules to a term until no more rules apply. However, this procedure is usually not adequate for program transformation. One rule may be the inverse of another, leading to non-termination, or different rule application orders may give different results (non-confluence). Stratego sidesteps these issues by allowing the programmer to declare the order of application using *programmable rewriting strategies* [51].

A strategy is an algorithm for traversing a term and applying selected rules in selected places in a selected order. The basic strategy is a single rule application, indicated by the name of the rule, which transforms a term at the root. When traversal comes into play 'the root' at which a strategy applies shifts. Therefore, in Stratego we use the notion of 'the current term' to indicate the (sub-) term at which the algorithm is currently focusing. Since a rule may fail to apply to a particular term (when the left-hand side does not match or the condition fails), a strategy may fail as well.

Rules are combined into more complex strategies by means of combinators. The fundamental combinators are sequential composition and deterministic choice. Sequential composition s_1; s_2, which applies first s_1 to the current term and then s_2 to the result. Deterministic choice $s_1 \Leftarrow s_2$ first tries to apply strategy s_1 and if that fails, applies strategy s_2. The basic strategies `id`, which always succeeds and `fail`, which always fails, are useful in combination with these combinators. Stratego programs can introduce new user-defined combinators through strategy definitions. For example, consider the following two definitions:

```
try(s)    = s ⇐ id
repeat(s) = try(s; repeat(s))
```

The `try(s)` strategy tries to apply a strategy `s` or else defaults to the identity strategy. The `repeat(s)` strategy repeatedly applies `s` to the current term until it is no longer applicable. Note that strategy definitions can be recursive. Furthermore, note that the current term the strategies apply to is implicit in these definitions.

In order to traverse a term and control where in a term transformations should be applied, Stratego provides *traversal combinators*. The basic idea underlying *generic* term traversal is the provision of *one-level* traversal combinators, which apply a strategy to the direct subterm of a term. The most prominent of these is the `all` combinator, which applies its argument strategy to all direct subterms of a term. It is typically used in definitions such as the following:

```
topdown(s) = s; all(topdown(s))
```

This definition introduces the generic traversal `topdown(s)`, which traverses an entire term and applies strategy `s` in pre-order to its subterms. If `s` fails at any point, so does `topdown`. `topdown(s <+ id)` will traverse the tree, apply `s` wherever possible and ignore any failures.

Rules are not really the atomic strategies of Stratego, they are rather strategy molecules, which are broken down into the atomic operations of matching a term against a pattern (`?t`) and instantiating (building) a pattern (`!t`). The Stratego compiler translates higher-level constructs such as rewrite rules into these lower level operations. However, they are also directly available to the Stratego programmer.

Stratego is not the only language which uses the concept of strategies. Visser's survey of strategies in rule-based transformation systems provides an overview of various approaches to programmable strategies [49].

2.5 Dynamic Rewrite Rules

Rewrite rules are *context-free*, i.e. only have access to the term to which they are applied. To express *context-sensitive* transformations, Stratego has introduced *dynamic rewrite rules* [14,34], which allow the definition of rewrite rules at run-time. Such rules can inherit information from the context in which they are defined and propagate this to the location where they are applied. For example, consider the following definition:

```
subs-assign = ?|[ x := e ]|; rules( Substitute : |[ x ]| -> |[ e ]| )
```

The `subs-assign` strategy first matches the current term against a term pattern for an assignment, binding the meta-variables `x` and `e`. Subsequently, a new dynamic rule `Substitute` is defined, rewriting the variable in the left-hand side of the assignment to the expression in its right-hand side. When the `Substitute` rule is later (in a different location in the term) applied to an occurrence of the variable `x`, it is replaced by the expression `e`. In addittion to the definition of new rules, Stratego supports a number of other operations on dynamic rules including the undefinition of rules, and limiting the scope of rule defintions. For a discussion of these features see [14,34].

2.6 Example

The features of Stratego are illustrated in Figure 2, which defines a flow-sensitive, intraprocedural constant propagation transformation for an imperative language with assignments and structured control flow, as illustrated by the transformation in Figure 3. The rewrite rules `EvalBinOp` and `EvalIf` express constant folding, that is, replace operator applications with (partially) constant arguments with the result of their evaluation. Typically, a constant propagation transformation will have a large number of such rules.

```
module tiger-propconst
imports Tiger
strategies

  main = io-wrap(propconst)

rules

  EvalBinOp : |[ i + j ]| -> |[ k ]| where <add>(i,j) => k

  EvalIf : |[ if 0 then e1 else e2 ]| -> |[ e2 ]|

strategies

  propconst =
    PropConst
    <+ propconst-assign
    <+ propconst-if
    <+ propconst-while
    <+ all(propconst); try(EvalBinOp <+ EvalIf)

  propconst-assign =
    |[ x := <propconst => e> ]|
    ; if <is-value> e
      then rules( PropConst :   |[ x ]| -> |[ e ]| )
      else rules( PropConst :- |[ x ]| ) end

  propconst-if =
    |[ if <propconst> then <id> else <id> ]|
    ; (EvalIf; propconst
        <+ (|[ if <id> then <propconst> else <id> ]|
            /PropConst\ |[ if <id> then <id> else <propconst> ]|))

  propconst-while =
    |[ while <id> do <id> ]|
    ; (|[ while <propconst> do <id> ]|; EvalWhile
        <+ /PropConst\* |[ while <propconst> do <propconst> ]|)
```

Fig. 2. Flow-sensitive constant propagation

```
(a := 1;                        (a := 1;
 if foo()                        if foo()
  then (b := a + 1)    ⇒         then (b := 2)
 else (b := 2; a := 3);         else (b := 2; a := 3);
 b := a + b)                     b := a + 2)
```
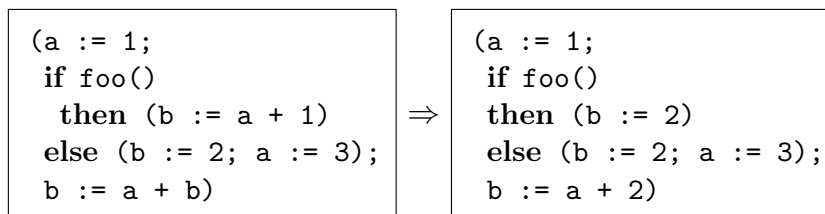
Fig. 3. Example of flow-sensitive constant propagation

The `propconst` strategy traverses the statements in a function body in order to realize constant propagation. The key operation is the application of the `PropConst` rule to replace variable occurrences by their statically constant value, if they have one. `PropConst` is a dynamic rule defined by the `propconst-assign` strategy as will be discussed shortly. If the current term is not a constant valued variable, the other alternatives of the `propconst` strategy are applied. The first three are special cases, which will be discussed shortly. The final case is the default, which applies a generic traversal combinator in order to apply `propconst` to all subterms. Eventually, this has the effect of a bottom-up traversal of the term. After this recursive traversal, an application of the evaluation rules is used to achieve constant folding. The other elements of the choice handle special cases. The `propconst-assign` strategy defines the dynamic rule `PropConst` to rewrite occurrences of the variable x to the constant right-hand side of the assignment. In case the expression is not a constant, the rule is undefined to prevent propagation of another value.

The `propconst-if` and `propconst-while` strategies define flow-sensitive propagation through control flow constructs [14,34]. The key combinator here is dynamic rule intersection s_1 `/R\` s_2, which applies the strategies s_1 and s_2 sequentially to the current term, each with a copy of the dynamic rule set for `R`. After applying both strategies, the resulting rulesets for `R` are intersected. In `propconst-if` this is used to apply the `propconst` strategy to the two branches of the `if-then-else` with the same initial ruleset.

2.7 Modules and Reuse

Stratego has a simple module system that allows programs to be divided into reusable chunks. Modules define signatures, rules, and strategies and can import other modules via an import clause (Figure 2). Modules can be organized into hierarchies. For example, Figure 4 shows the structure of the Stratego Library. Version 0.17 does not support hiding of rules and strategies (other than using local let bindings). Namespace management is planned for one of the next releases.

Modules provide a coarse grained method for dividing a program into files. Reuse with finer granularity is provided by the separation of rules and strategies. Modules can be used to provide collections of useful rules and strategies, which can be combined in many different compositions. Since rules are named, they can be selected separately in any transformation. Generic strategies capture a particular transformation strategy that can be instantiated with appropriate transformation rules. For example, the constant propagation strategy in Figure 2 can be refactored into a generic forward data-flow transformation strategy, parameterized with the specifics of a transformation [34].
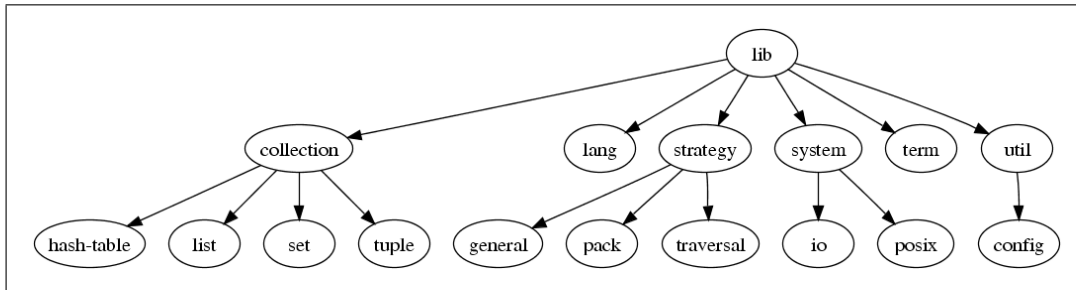
Fig. 4. Structure of packages in the Stratego/XT library.

2.8 Stratego Compiler

The primary method for running Stratego programs is via compilation. The Stratego Compiler `strc` translates Stratego programs to C programs and uses `gcc` to compile these to executable programs. The C programs generated by `strc` rely on the ATerm library [36] for the internal and external representation of terms, and for garbage collection. `strc` is a whole program compiler, which entails that all imported modules are read and translated to C. To reduce compile-time and the size of compiled programs, Stratego supports a form of separate compilation through the use of *external definitions*, which declare strategies that are implemented in a separately linked library. The compiler has a separate mode for compiling a collection of modules to a shared library and a module with external definitions to be imported by client applications (or libraries).

2.9 Stratego Shell

Compilation is the normal mode for development of transformation systems with Stratego. Indeed, we usually do not invoke the compiler from the command-line 'by hand', but have an automated build system based on (auto)make to build all programs in a project at once. For learning to use the language this can be somewhat laborious. Therefore, we have also developed the Stratego Shell, an interactive interpreter for the Stratego language. The shell allows users to type in transformation strategies on the command-line and directly see their effect on the current term. While this does not scale to developing large programs, it can be instructive to experiment while learning the language.

2.10 Stratego Library

The Stratego Library was designed to contain a good collection of strategies, rules and data types for manipulating programs. However, the library also defines standard data types, such as lists, strings, hashtables, sets, file and console I/O, directory manipulation and more. The organization of the Stratego Library is hierarchical. At the coarsest level of organization, it is divided into packages, named with paths such as `collection/list`. Each package in

turn consists of one or several modules. A module is a leaf in the hierarchy. It maps to one Stratego (.str) file, and contains definitions for strategies, rules, constructors, and overlays.

Figure 4 shows the package structure of the Stratego Library. The `collection` package implements standard collection data structures. The `lang` package provides support for high-level constructs which are translated by the compiler to lower-level constructs in combination with library calls. For example, dynamic rule definitions and invocations are translated to regular Stratego code with calls to the dynamic rules library. The `strategy` package implements a large number of generic strategies, including a large library of standard traversals. The `system` package implements the interface to external system functionality such as input/output and process management. The `term` package provides strategies for generic manipulation of terms. Finally, the `util` package provides assorted utilities.

The library consists of over 10K lines of code divided over more than 60 modules defining over 1000 strategies and 300 rules, with an additional 4K lines of code for unit tests. API documentation is automatically generated from the sources of the library by xDoc [3] and browsable online. The Stratego Library is available to programmers as a separately compiled shared library. In addition to the base library, Stratego/XT makes available several other libraries implementing infrastructural functionality for transformation systems. These libraries are the basis of the XT transformation components discussed in the next section.

3   The XT Transformation Tools

In the previous section we discussed the Stratego language used to implement transformations on (abstract syntax) trees. XT complements Stratego with a set of small languages and tools needed to realize the other aspects of transformation systems [27]. Each language and supporting tool set targets one clearly defined task, and is used to build components which compose with each other to form a complete transformation system.

3.1   Syntax Definition

Syntax definitions play a central role in XT, as they are used to specify the syntax of programming languages in a declarative way, constituting the primary method for defining the structure of the data transformed by Stratego

---

[3] xDoc is a generic documentation generation system implemented in Stratego [41]. The tool is not part of the Stratego/XT distribution.

programs. Several code generators found in XT take the syntax definition as input, deriving multiple artifacts from the same definition. Figure 5 illustrates how various XT tools are used to derive (1) a parser which directly constructs an AST from a source code file, (2) a Stratego signature (data declaration) for the AST, (3) a format checker for such ASTs (used to determine the correctness of subsequent transformations on the AST), and (4) a pretty-printer for turning ASTs back into text. The `Parser` and `Pretty-Printer` in the pipeline of Figure 5 are entirely derived from the SDF definition. The `Transform` component (which may be a series of components), is typically written in the Stratego language.

The XT collection uses the syntax definition formalism SDF [42]. The SDF parsing technology provides a parser generator and the scannerless, generalized-LR parser SGLR [42,39] [4]. SDF is different from most other grammar formalism in that it is highly *modular* and *declarative*. This allows the formalism to easily scale to large language declarations, but more interestingly, it also allows for easy language composition and embedding (see Section 5 for a discussion). The following module illustrates a few of the features of SDF:

```
module Expressions
exports
  sorts Id Exp
  lexical syntax
    [A-Za-z][A-Za-z0-9]* -> Id
  context-free syntax
    Id                                -> Exp {cons("Var")}
    Exp "+" Exp                       -> Exp {cons("Plus"),left}
    "if" Exp "then" Exp "else" Exp "end" -> Exp {cons("If")}
```

First, the language is modular, that is, syntax definitions can be divided into separate modules, which may be reused independently. Next, the definition of lexical and context-free syntax are integrated into a single formalism. Finally, constructor annotations on productions are used to define the mapping from concrete syntax trees to abstract syntax trees.

The XT collection extends the SDF tool set with a small, declarative language for unit testing syntax definitions, called ParseUnit. For example, the following ParseUnit specification defines a test for the expression language defined above.

```
testsuite Expressions
options topsort Exp
test simple addition
  "x + y" -> Plus(Var("x"),Var("y"))
```

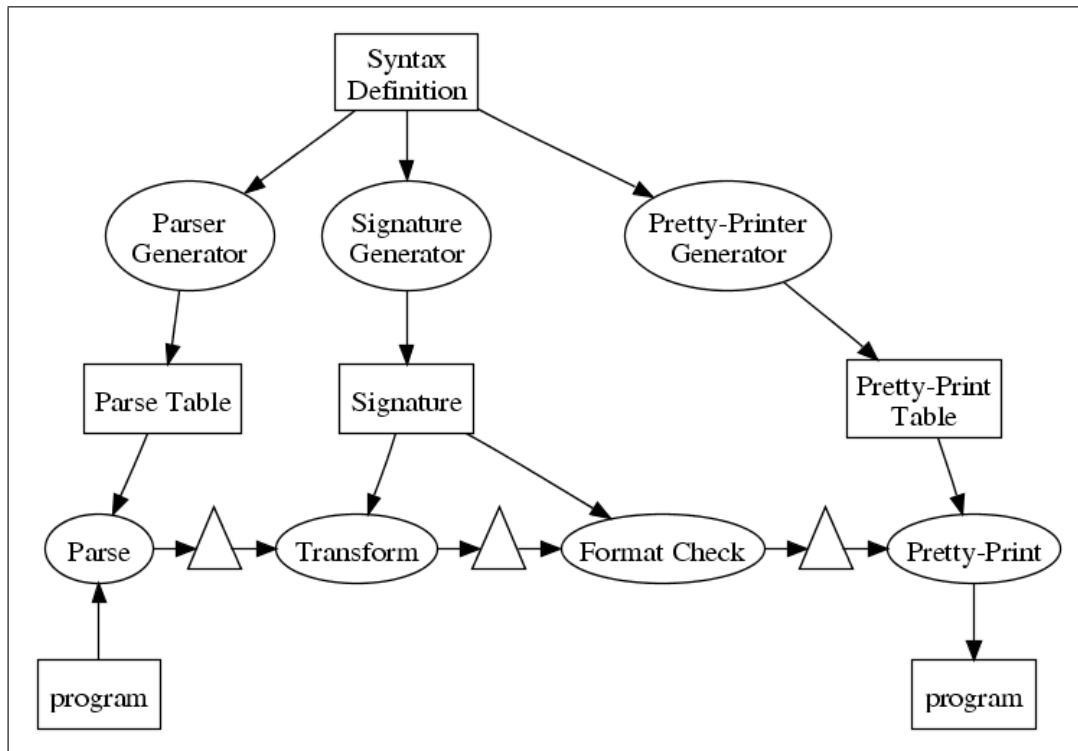[4] SDF and SGLR are developed at the CWI in Amsterdam, The Netherlands.

Fig. 5. Transformation infrastructure

The test defines the input string and the abstract syntax tree that should result from parsing it.

3.2 Exchange Formats

The components in XT exchange data using one common format, Annotated Terms, or ATerms [36]. ATerms are used to represent the parse trees produced by the SGLR parser [5], as well as the abstract syntax trees exchanged between components written in Stratego. In normal operation, the terms are exchanged and stored in compressed binary form, but they can be converted to a textual representation which is readable to humans. Additionally, the XT collection contains tools which can convert XML documents to ATerms and vice versa, which allows for interoperation with external tools.

3.3 Pretty-Printing

We refer to the conversion of ASTs back into source code as *pretty-printing*. In principle, pretty-printing is the inverse of parsing, and the construction of pretty-printers starts from the syntax definition. Pretty-printing consists of

---

[5] Parse trees contain information, such as whitespace, comments, and references to the grammar productions, that is not present in abstract syntax trees. Stratego programs can manipulate any form of ATerm, including parse trees. Indeed, the transformation from parse tree to abstract syntax tree is expressed in Stratego.

two stages. In the first stage an abstract syntax tree is converted to a declarative layout in the Box language [19]. The second stage is the formatting of Box expressions to text (or some other display format such as HTML or LATEX). This last stage is implemented by the standard `box2text` XT component. The first stage can be expressed in plain Stratego, or by means of a pretty-printing table, effectively a DSL for pretty-printing. For example, the following rules define a pretty-printer for the `Expressions` language:

```
[ Var  -- _1,
  Plus -- H[_1 KW["+"] _2],
  If   -- V[ V is=2 [ H[KW["if"] _1 KW["then"]] _2 ]
             V is=2 [ KW["else"] _3 ]
             KW["end"] ] ]
```

For each constructor in the abstract syntax a mapping to a Box expression is defined, using numbered placeholders such as `_1` to refer to arguments of the constructor. Typical Box operators are `H` for horizontal composition, `V` for vertical composition, and `KW` for keywords.

Figure 5 indicates that a basic pretty-print table can be derived from a syntax definition. In order to achieve a *pretty* result, the rules need to be adjusted; it is a rather hard problem to automatically create a set of formatting rules that is considered to be pretty by humans. When the syntax definition is changed, a new pretty-printer must be generated. In order to reduce the maintenance problem this causes, the new basic pretty-print table can be automatically merged with the old adjusted pretty-print table, in order to keep all original rules that are not affected by the changes.

The declarative nature of SDF is also helpful in other aspects of pretty-printing. Priorities of operators are declared explicitly, instead of encoded into the grammar productions. By using the parenthesizer generator in XT, accurate rules for correctly parenthesizing source code text from ASTs can be automatically generated. As a result the pretty-printed source code contains the minimal number of parentheses needed to preserve the semantics (operator ordering) of the AST.

As noted above, abstract syntax trees normally do not preserve whitespace and comments. For applications in which source code is transformed and returned to the programmer (e.g. refactorings), layout preservation is generally important. The XT tools provide support for layout preservation. However, an inherent problem of any syntactic or semantic annotations to abstract syntax trees, is that their preservation throughout transformations requires attention in (potentially) all transformation rules.

Another issue in pretty-printing is the placement of parentheses. While a conservative pretty-printer can avoid the issue by adding excess, parentheses this

leads to results that are not very readable. For example, the parentheses in $(x+(3*y))$ are not necessary. On the other hand, the parentheses in $(x+3)*y$ cannot be removed without changing the meaning of the expression. Parentheses are removed from the abstract syntax tree representation, since they are superfluous in a structured representation. Therefore, they need to be reintroduced, only where needed, prior to translation to a Box expression. The `sdf2parenthesize` tool generates from a syntax definition a transformation that introduces the required parentheses to an abstract syntax tree.

3.4   Format Checking

Syntax definitions are annotated with instructions for constructing abstract syntax trees from parse trees (through constructor annotations). The syntax definition is therefore also a declaration of all valid abstract syntax trees. When ASTs are later transformed, it is often useful to check if they are still structurally valid. The validity of tree structures can be described using regular tree grammars [10]. A regular tree grammar describes context-free constraints over tree structures. For example, the following regular tree grammar describes well-formed abstract syntax trees corresponding to the `Expressions` language defined above:

```
regular tree grammar
  start Exp
  productions
    Exp -> If(Exp,Exp,Exp)
    Exp -> Plus(Exp,Exp)
    Exp -> Var(Id)
    Id  -> <string>
```

XT provides tools for deriving a regular tree grammar from a syntax definition, and a format checker which checks if a given ATerm conforms to such a regular tree grammar. Signatures for use in Stratego programs as described in the previous section are in turn derived automatically from regular tree grammars.

3.5   Component Composition

A complete transformation system consists of a (potentially large) number of components, including parsers and pretty-printers. Different transformations for the same programming language may share components. For example, Figure 6 shows a data-flow diagram for a Tiger interpreter and partial evaluator, which share several front-end components. Component composition can be realized in two ways. The preferred approach in current Stratego/XT is to define components as separately compiled shared libraries providing a collection of transformation strategies. Such libraries can be combined to form an application, which basically consists of a pipeline of calls to library strategies, and a command-line interface (for which the Stratego library provides support).
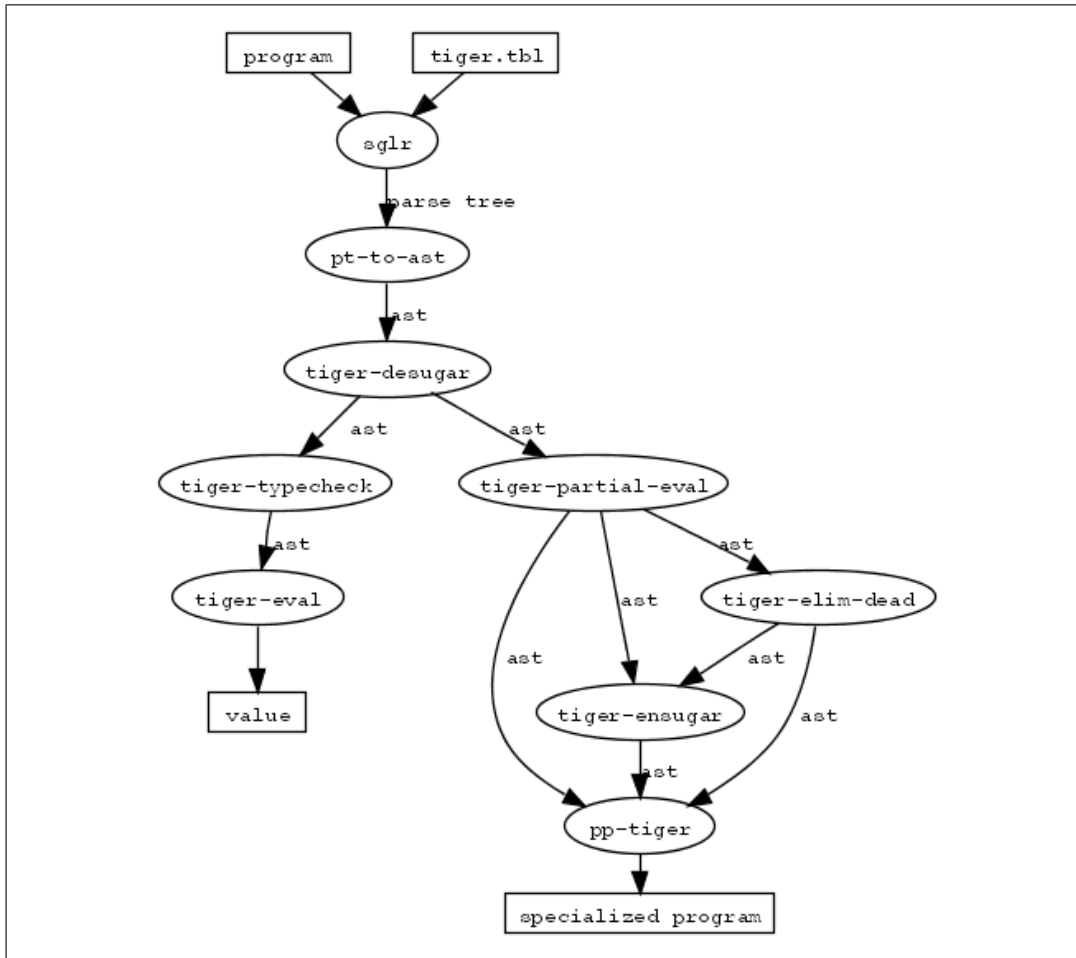
Fig. 6. Composition of transformation tools into a transformation system.

Before separate compilation was available in Stratego/XT, a different approach to component composition was used, which may still be applicable in some cases. In the transformation tool composition (XTC) approach, components are compiled as stand-alone transformation tools (executables) that are run as separate processes. Communication between tools is realized seamlessly using the ATerm exchange format. The XTC library provides support for calling a tool as a separate process and arranging the exchange of terms with the called tool. The approach has two main problems. The first is that calling components as separate processes requires knowledge of the location of the executable. (To some extent this problem also exists for shared libraries, but the operating system infrastructure appears to be better organized for this scenario.) XTC mitigates this problem by parameterizing a tool with a repository of tool locations, such that no absolute path names need to be embedded in programs. Furthermore, the marshalling and unmarshalling of data for exchange is expensive. Despite these disadvantages, the approach may still be valuable in situations where third-party tools not link-compatible with Stratego programs need to be integrated in a transformation system. In Stratego/XT 0.17 most uses of XTC have been replaced with shared libraries.

4   Implementation

Stratego and the XT tools are bootstrapped, that is, they are used in their own implementation. The 0.17 release of Stratego/XT containss well over 50K lines of Stratego code. In this section we reflect on the implementation techniques and software engineering aspects of the project.

4.1   Domain-Specific Languages

Many of the tools and languages found in XT have been developed using Stratego/XT. For example, format checkers are expressed in a regular tree grammar (RTG) language whose syntax is defined using SDF. The interpreter for this language is implemented in Stratego, and makes heavy use of dynamic rules for run-time generation of evaluation rules. The same story holds for the Box language and its interpreter. Not all the small languages are interpreted, however. In the case of `sdf2parenthesize`, the tool which generates a parenthesizer for ASTs, the final product is always a stand-alone, executable program. `sdf2parenthesize` will generate a Stratego program that, after compilation, can be applied to an AST and produce another AST with proper parentheses added.

4.2   Bootstrapping the Stratego Compiler

The Stratego Compiler itself is also bootstrapped. The very first Stratego compiler was written in SML. That compiler was then used to compile a rewrite of the compiler in Stratego. Since then the compilation of Stratego programs to C is implemented as a series of transformations in Stratego. The syntax of Stratego is defined in SDF. Bootstrapping has proven to be a good approach for developing the compiler and the language because it provides a realistic case study and a good test case for the compiler. The source of the compiler comprises some 90 Stratego modules with a total size of some 9K lines of code. Additionally, over 140 test programs comprising 6K LOC are used to test the compiler.

The overall architecture of the compiler follows the traditional organization of a compiler into a front-end, optimizer, and back-end (Figure 7). These components are organized as a sequence of separate 'source-to-source' transformation tools. Note that the there are multiple paths through the data-flow graph of the archicture, indicating that some transformations are optional; these are applied with higher levels of optimization. The largest part of compilation consists of transformations from Stratego code to Stratego code. Only the last transformation `s2c` translates Stratego Core programs to C code.
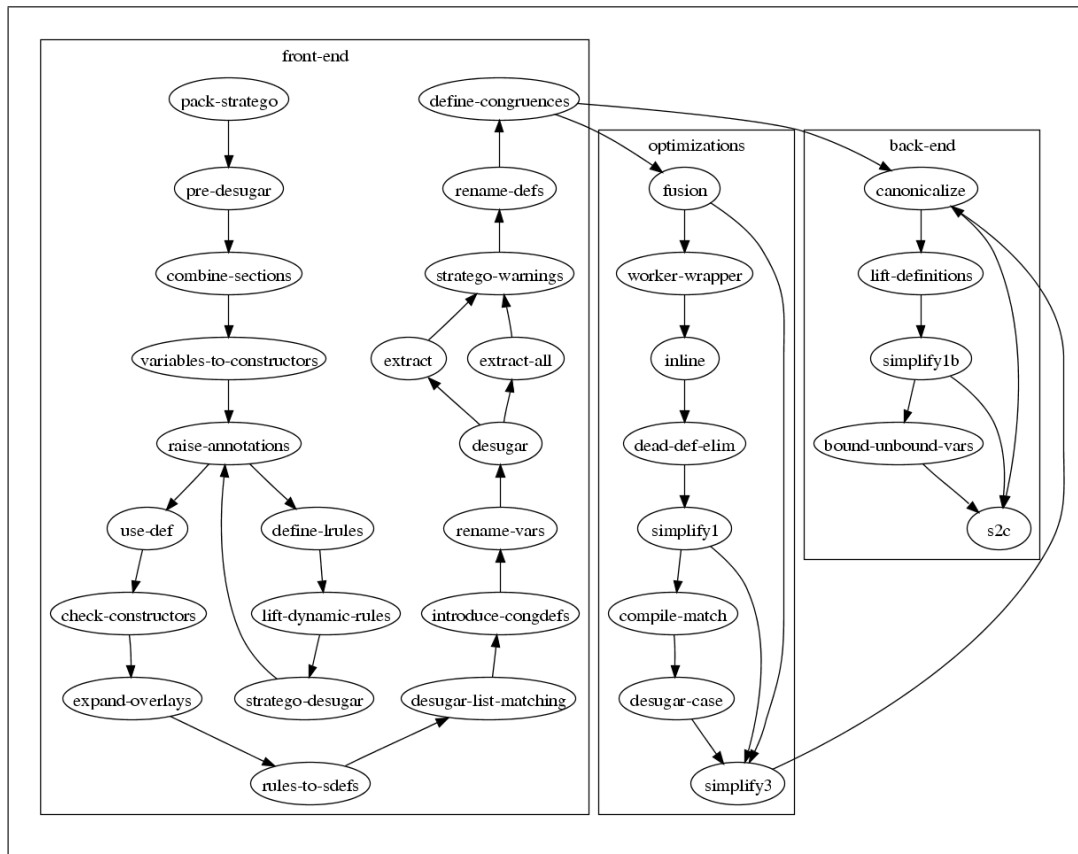
Fig. 7. Architecture of the Stratego Compiler

4.2.1 Front-end

The front-end of the compiler collects the code for all modules of a program, performs a number of static analyses, and desugars the program to Stratego Core. The static analyses comprise checking whether constructors are declared, whether variables are bound before being used in a build, whether invoked rules and strategies are defined, and some properties of dynamic rules. Stratego is not a typed language, so typechecking is not performed. Desugaring takes place in several stages and involves simplifying programs from the full Stratego language to the Stratego Core language, a minimal subset of the full language. This involves various basic simplifications, such as the merging of multiple definitions with the same name, but also a complex translation of Stratego with dynamic rules into plain Stratego. Since at various stages of compilation different subsets of full Stratego are expected, format checking against subsets of the full regular tree grammar is used to monitor the integrity of the compiler, especially during development and testing of the compiler. Complete format checking can be turned on at all stages of compilation in the compiler, however, by default this is only done at a few crucial stages of compilation.

4.2.2   Optimizations

Before application of the back-end, several optimizations may be applied to Stratego Core programs. A collection of simplification rules based on algebraic laws of the Stratego language are applied to all strategy expressions. Innermost fusion is a transformation optimizing instantiations of the `innermost` strategy producing super-linear speed-ups [25,26]. The optimization solves the problem of re-normalizing terms already in normal form without the overhead of tagging or memoizing terms. Inlining of strategy definitions enables simplification across calls, and dead definition elimination removes definitions no longer used. A number of other optimizations that used to be part of the compiler is currently under revision.

4.2.3   Back-end

The back-end of the compiler translates Stratego Core programs to C. The translation scheme used in the back-end has evolved from the generation of virtual machine code instructions with a dedicated stack to generation of idiomatic C code. From the very first version, the generated code has depended on the ATerm library for the internal and external representation of terms and for (conservative) garbage collection [9]. This library has in fact made the development of the Stratego Compiler possible; development of a dedicated run-time system would have been a major undertaking. Challenging issues in the compilation of Stratego are the treatment of failure in strategies and dealing with bindings to variables in non-local scopes.

Stratego's choice operator implements *local choice*. That is, in the strategy expression s_1 `<+` s_2, if s_1 fails, computation backtracks and applies s_2. However, once s_1 has succeeded, the choice is committed and the contintuation of the expression never backtracks to s_2. Early on in the developmen of Stratego, it was established that the alternative of global backtracking was rarely needed and much more expensive to implement. In the translation scheme introduced in Stratego 0.6, setjmp/longjmp, the built-in *exception handling* mechanism of C, was introduced to handle *failure and choice*. This mechanism allows a clean translation, since it abstracts from the administration of choice points. However, since the control-flow in Stratego programs depends heavily on failure and choice, the overhead of setjmp/longjmp is significant. Especially on architectures with large register sets (such as the PowerPC used by Mac OS X) the cost of saving register windows has meant a serious performance penalty. In Stratego/XT 0.16, a new translation scheme has been adopted, representing failure as a `NULL` ATerm pointer. On return of a strategy expression the caller needs to determine by means of a comparison with `NULL` if the strategy was a success or failure, in constrast to the setjmp/longjmp solution, which entails a direct jump from the failure point to the choice point in the code. While the latter approach would seem to be more efficient than the explicit

unwinding of the stack, the opposite turns out to be true. The explicit failure representation provide a performance boost, especially on Mac OS X.

Stratego supports nested strategy definitions. Even though local definitions are only used sparingly in actual Stratego code, the transformations in the compiler introduce many local functions to encapsulate strategy expressions passed as arguments to other strategies. Starting with Stratego 0.6, nested definitions were implemented using a non-standard feature of C, supported by the GCC compiler, namely *nested functions*. With this feature, the implementation of closures comes for free, i.e. is delegated to the C compiler. While convenient for code generation, use of nested functions entails a dependency on GCC. Furthermore, GCC's implementation of nested functions based on *trampolines* requires execution of instructions stored on the stack. For safety reasons, more and more platforms forbid execution of code on the stack (by default) entailing reduced portability of Stratego programs. Therefore, in Stratego 0.17, the implementation of nested definitions is changed to explicitly implement closures in the back-end of the Stratego compiler. As a result, the compiler generates ANSI C compliant code, no longer depending on gcc.

While the Stratego compiler is a whole program compiler, it also supports a form of *separate compilation*. A collection of modules can be compiled into a shared library and an interface declaration in the form of a module with *external definitions*. Client programs can use such a library by importing the external definitions and linking against the shared library. This approach is used for the Stratego Library, and makes a big difference in compile time.

4.3 Software Development Process

Stratego/XT is developed in the open. This means first of all that the software is open source; the source code for Stratego/XT is distributed under the GNU LGPL license. This license allows the development of closed source (commercial) transformation systems based on Stratego/XT. Furthermore, the *development process* is open, i.e. both the services used in the development, such as the issue tracker, mailing lists, wiki and build farm and the source code itself are visible to everyone all the time. The openness of the development process has fostered a small community around Stratego/XT, and patches from external contributors can be found in almost every release.

4.3.1 Quality Management

Testing is the corner stone of the automated quality assurance process for Stratego/XT. Whenever a new strategy, module or rule is added to the Stratego library or a new component is added to the XT collection, unit tests are also added. These unit tests ensure that the intended semantics of the addition is kept in future revisions. Unit tests are also written for syntax definitions us-

ing the Parse Unit language. The compiler is tested using a large collection of test programs. Whenever a bug is detected and fixed, new unit tests are added to avoid regression as development continues. These unit tests are exercised by the build farm as part of the continuous integration process, described later.

4.3.2   Version and Issue Management

The source code for Stratego/XT and many of the dependent projects is managed using the Subversion version management system, which allows unique identification of all files in a revision and refactoring of the source tree. The Subversion repository is world readable, allowing anyone to do a checkout of the latest version. Trusted developers can get write access to the repository. Figure 8 illustrates the activity in the Stratego/XT by the number of commits per month. Figure 9 shows the development of the size of the distribution in terms of lines of Stratego code; the steady growth of the sources has been reversed in recent distributions by a drastic dead code removal and refactoring operation.

Bug reports, feature requests, and internal development tasks for Stratego/XT and related projects are managed using the web-based JIRA issue management system. All reported issues are scheduled into specific future releases. In addition to providing an easy way for Stratego/XT users to report bugs, the tracker is also an excellent way for interested parties to follow the direction of development.

4.3.3   Continuous Integration and Automatic Release Management

For release management and continuous integration, the Stratego/XT project uses a build farm based on the Nix software deployment system [20]. The build farm produces releases, performs integration testing, and verifies the portability of the software for several platforms.

Continuous Integration. The build farm monitors the Subversion repositories of the various Stratego/XT packages and starts building a package on several platforms if there has been a commit in a package or one of its dependencies. The build farm performs a full check of the package and reports errors to subscribers by email and on the web. These timely and automatic rebuilds are crucial for catching programming errors and bugs as early as possible. Early detection, in turn, makes the errors easier to understand and correct. The fully automated testing is in particular useful for testing if the Stratego compiler can be bootstrapped against itself. We only upgrade the bootstrap Stratego compiler if the build farm has guaranteed that this actually works.

Portability Testing. The Nix build farm is also used to compile Stratego/XT on different operating systems and platforms, currently being Linux, Mac OS X, and FreeBSD. Continuously testing of portability has proven to be very
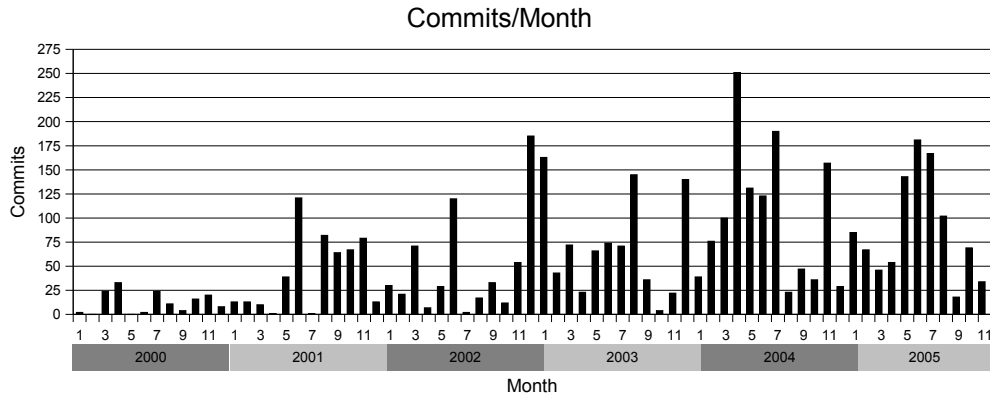
Commits/Month

Fig. 8. Distribution of number of commits in the Stratego/XT source tree over the last six years; 17 contributors committed 4255 times. (Not counting commits in related projects in the same repository.)
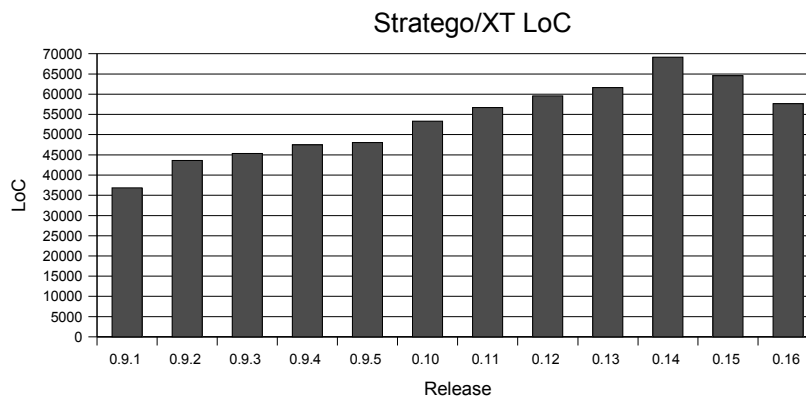
Stratego/XT LoC

Fig. 9. Size of subsequent Stratego/XT distributions (not including generated C sources).

useful, since minor portability issues, such as differences in the implementation of dynamic libraries, non-portable shell scripting, and use of non-standard functions are caught very early in the development process and can be fixed when the developer is still working on that issue.

Continuous Release. The Nix build farm not only builds the software, but also continously creates source and binary distributions. In this way, the distribution process itself is tested as well (e.g. missing files in a distribution or incomplete RPM dependencies) and users of Stratego/XT can easily obtain the latest developments, if necessary, by just downloading a source or binary distribution from the Nix build farm. This approach of continuous release has been very fruitful, since we receive feedback from users even before the software is actually released with a stable version number. Also, users do not have to suffer the problems in building from Subversion.

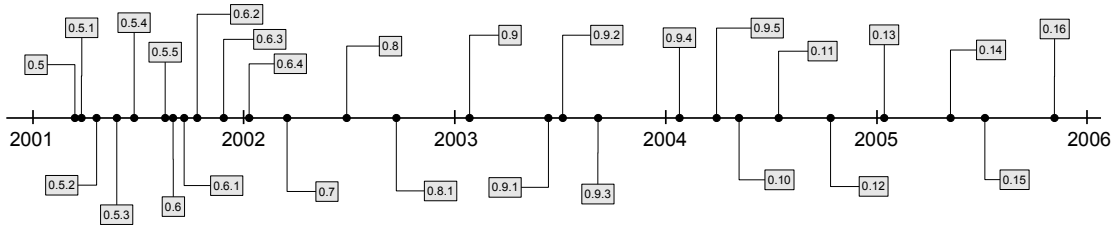*Release Management* Actually, the Nix build farm completely manages the

Fig. 10. Release History of Stratego/XT

release process of Stratego/XT. For releases, we create branches in the Subversion repository, which are tagged as stable. After this, the Nix build farm will automatically produce a tested source distribution, RPMs for several Linux distributions, and Nix packages for installation with the Nix deployment system itself. This automation of the release process, helps a lot in the steady production of new releases. Figure 10 shows that a new major release of Stratego/XT is released about 4 times a year.

Baseline Development. Several times during the development of a new Stratego/XT release, specific revisions are elevated to a new 'baseline'. Both Stratego/XT and the dependent projects are always built against the latest baseline. Thus, promoting a revision from the source code repository to a new baseline requires testing that release against a substantial number of projects, catching regressions which are otherwise difficult to catch with unit tests alone.

4.3.4   Documentation

The primary source of documentation is the manual. It offers an extensive introduction of the XT architecture, and also a complete Stratego tutorial. The tutorial includes several program transformation examples shown on a small, imperative language. The reference material includes complete manual pages for all the XT command-line tools, and online API documentation of the library, which is also available for download.

4.3.5   User Support

The Stratego/XT website [52] contains pointers to mailing lists for users and developers, a wiki, release pages, documentation, build farm and the issue tracker. For live chats with the developers, join the IRC channel `#stratego` on `irc.freenode.net`.

5   Experience

Stratego/XT is being applied in a number of research and industrial projects. The experience from these projects has been influential on the design and

implementation of Stratego/XT.

5.1 Extensions of Stratego/XT

As Stratego/XT is developed in Stratego/XT, extending the environment is quite easy. As a result, many extensions and utilities have been built using Stratego/XT, which complement and extend the development environment. In the utilities category, perhaps the most important of examples are the Stratego Shell and the xDoc system. Stratego Shell is an interactive interpreter for Stratego that can also execute Stratego programs as if they were scripts. It has been used in education, and is also a very convenient tool for quickly testing code snippets and XTC compositions during development. xDoc, is a Javadoc-like source code documentation system in the style of Javadoc, but for Stratego code [41]. The API reference documentation for all releases of Stratego/XT have been generated using xDoc. An example of a language extension of Stratego/XT is Aspect Stratego [29], which adds aspect-oriented features to Stratego. The extension provides Stratego with new language primitives for specifying pointcuts and advice that are weaved into the code at compilation time.

5.2 Tiger

Andrew Appel's compiler textbook example language Tiger [1], makes a perfect playground for experimenting with program transformation. The language is small, which makes it manageable, yet it is not trivial and includes nested functions, arrays and records, which makes it interesting. For a course on Program Transformation and a seminar on High-Performance Compilers, we have evolved a complete compiler for Tiger that includes all aspects of compilation, from type checking, via optimizations, to instruction selection. Many innovations in Stratego/XT were first tested using Tiger [44,16,46,14,34]. The Tiger-Base package is a subset of the full Tiger compiler package with only the syntax definition and source-to-source transformations on Tiger programs.

5.3 BibTEX

The Stratego/XT BibTEX Tools package provides a collection of transformations on BibTEX bibliography files [48]. The main application is a transformation from BibTEX to sectioned bibliographies (publication lists) in HTML and PDF, which involves external tools such as `latex`, `bibtex`, and `hevea`, a LaTEX to HTML translator. The transformations include an interpreter for a little query language for selecting entries from a file, and replacing of queries embedded in a LaTEX document with citations of the selected entries. The package was developed for the automatic production of online publication lists using different organizations (e.g. by year, by type) and links. It is used for maintenance of several such lists, including the publication lists on the

Stratego website. It provides a nice example of application of Stratego/XT to document transformation.

5.4 Java

For Java, we have developed a modular Java 1.5 syntax definition, a high-quality pretty-printer, a name resolution phase, an extensible type checker (including generics), an extensive reflection library for use in Stratego, and tools for reading and writing Java bytecode to terms. These tools are used in the implementation of *language embeddings*, a recently added application area of Stratego/XT [17,11]. Typical case studies of language embedding are extensions of Java with domain-specific languages for user-interfaces and regular expressions are available. JavaJava [15] is an advanced code generation tool, based on the extensible type checker and the GLR parsing technology used in Stratego/XT. Also, we have developed an AspectJ grammar, with is a modular extension of the Java syntax definition [13]. Thus, the AspectJ grammar only defines the syntax extensions to Java provided by AspectJ, and is programmed against the public interface of the Java grammar. This sort of language composition is possible because of the scannerless, GLR nature of the SGLR parser.

5.5 C/C++

To date, three frameworks for transforming C++ (and C) code are in construction. The Transformers project at LRDE, EPITA, France has produced a disambiguating front-end for C99, and has come close to finishing a similar front-end for ISO C++ 2003. Disambiguation of both languages requires semantic analysis. Disambiguation rules are implemented using an attribute grammar system, which is an extension to SDF. This extension, with an attribute grammar evaluator, has been implemented with Stratego/XT.

CodeBoost [2] is a source-to-source optimizer for C++ code, developed at the University of Bergen, Norway. Its main purpose is to provide a framework for implementing domain-specific abstractions with optimizations for numerical software. The parser is reused from the OpenC++ project, while the semantic analyser, which covers substantial parts of C++, is written entirely in Stratego.

The Proteus [54] project at Lucent, USA, has built a C/C++ transformation framework based on Stratego. The syntax is defined in SDF. Transformations are written in a high-level transformation language, YATL, and compiled to Stratego. Proteus uses yet another C++ frontend, which allows it to retain source code layout and also deal better with C++ pre-processor directives.

5.6 Miscellaneous

Stratego/XT has been used to build several other compilers and frontends.

- OctaveC is a compiler for Octave, a clone of Matlab. It includes loop vectorization, and partial evaluation [33].
- Prolog Tools provides a language front-end and DSL embedding for Prolog [23].
- Spoofax [28] is an editor for Stratego for the Eclipse IDE. It features content-aware, syntax highlighting editors for SDF and Stratego with outlines and content assistance.
- WebDSL [50] is a domain-specific language for dynamic web applications with a rich domain model.

6 Discussion

6.1 Previous Work

This paper and the Stratego/XT 0.17 release mark a milestone in the development of Stratego/XT, with the introduction of radically improved documentation and a robust release process implemented with the Nix system [20]. Compared to earlier descriptions of Stratego/XT [27,45,47,12] we have gained new experience with the development of transformation systems for Java, C, Octave, and BibTeX. Based on the feedback from these projects new language constructs such as dynamic rules and concrete object syntax have matured. Stratego/XT now also provides new tools for testing, validating, and debugging, to help in developing reliable transformation systems.

6.2 Related Work

Stratego/XT lives in a larger ecosphere of program transformation systems, many of which have provided inspiration for features found in Stratego, and some have also provided components for XT.

The closest sibling of Stratego/XT is ASF+SDF [38,40]. This is where the syntax definition language SDF discussed in Section 3 originates. ASF is a declarative language for specifying semantics of programming languages as algebraic rewrite rules. These conditional rewrite rules are applied to terms produced from source code using SDF, either by a rule interpreter or after compilation to binary code. ASF+SDF does not provide programmable strategies, but the overhead of traversals can be reduced by means of traversal functions [37]. Both the ASF and SDF language are supported by the MetaEnvironment, an interactive environment for developing, debugging and executing program

transformation systems built with ASF+SDF. In constrast, Stratego/XT only provides a programming environment based on command-line tools.

The ideas for the strategy operators in Stratego were inspired by the ELAN rewrite system [6], which introduced programmable rewriting strategies and congruence operators for term traversal. Generic traversal and dynamic rules are contributions of Stratego [32,51,44].

TXL [18] is a programming language for writing software analysis and source transformation tools, based on the tree rewriting formalism. The language is a hybrid of the functional and rule-based paradigms, which provides unification, implied iteration and deep pattern matching. Both the language grammar and transformation rules on programs conforming to the grammar are written in TXL. TXL provides a fixed set of traversals and does not support the definition of generic strategies.

The Design Maintenance System, DMS [5,4], is a commercially available collection of tools for writing custom program transformation systems. This tool collection contains domain-specific languages for writing transformations and grammars. From the grammars, GLR parsers and multi-pass attribute evaluators are generated. The transformations are executed by a rewrite engine, and controlled by a meta-language called XCL. The system comes with a large collection of ready-to-use language environments.

FermaT [56,55] is an industrial-strength formal transformation engine developed in the course of twenty years at Durham University, Software Migrations Ltd and De Montfort University primarily by Martin Ward. It uses formally proven program transformations which preserve or refine the semantics of a program while changing its form. In the context of FermaT, the purpose of such transformations is to restructure or simplify legacy systems, and to extract higher-level representations. The extracted representation is guaranteed to be equivalent to the original code.

TAMPR [7,8] is one of the first program transformation systems, dating back to the early 70s. It supports program transformation by means of rewrite rules. A set of rewrite rules is used to transform the abstract syntax tree of a program to canonical form by exhaustive application. A number of such canonicalizations can be combined by sequential composition. This sequential composition provides a simple mechanism for control over the application of rules, which can be seen as a subset of the combinators found in Stratego. The system has been applied to the derivation of a number of numerical software packages, including LINPACK.

Many of the tasks performed by program transformation systems are also solvable by other approaches, such as attribute grammar systems. JastAdd [22] is an extensible compiler construction framework based on modern attribute

grammars featuring reference attributes, equations and rewrite rules. At the heart of JastAdd is a modular, declarative attribute grammar language which is used to express attribute computations on ASTs. These ASTs must be produced by external parsers, as JastAdd does not come with a parser formalism of its own. The attribute grammar language allows normalizing rewrites and desugaring to be expressed directly, but more substantial rewrites must be written in Java.

More traditionally, compiler constructor suites have been around for decades, providing frameworks and infrastructure for construcing compilers and support tools. Eli [24] is a domain-specific programming environment, mainly aimed at constructing compilers, but which has been used for many program transformation tasks. The development of compilers in Eli centers around specifications of program transformations. The specifications include information about the source language syntax, source program tree, target program tree and machine instruction set. The designer of a particular transformation specifies an instance of the general compilation problem. Given a transformation specification, the Eli environment selects applicable tools which are used to derive executable components. These generated components are C programs which are integrated with standard components of Eli to produce a stand-alone program transformation system.

Extended parsing frameworks also have significant overlaps with program transformation systems in the tasks they are used for. ANTLR [35] is a popular framework for constructing parsers, compilers and transformers of formal languages, using Java, C#, C++ or Python as implementation languages. It comes with libraries for the implementation languages which are used for parse tree traversal, tree construction and transformation. ANTLR is centered around a an LL(k) parser generator, which gives it good error reporting capabilities.

Finally, Stratego has provided inspiration for other systems. In particular, the approach to strategy combinators and generic traversals has been adopted in a number of other languages. Strafunski is an implementation of generic traversal strategies in Haskell [31]. This work has been continued in a series of 'scrap your boilerplate' papers refining the implementation of generic programming in Haskell [30]. JJTraveler is a Java library implementing a collection of *visitor combinators* based on the basic Stratego strategy combinators [53]. Tom is an extension of Java with support for rewrite rules and rewriting strategies in the style of Stratego [3].

7 Conclusion

The goal of Stratego/XT is to support a wide-range of transformations and to provide a new level of abstraction for the implementation of transformation systems by third parties. In the last couple of years Stratego/XT has considerably matured due to intensive development and research. We have been successful in exploring the implementation of individual transformations, and the range of transformations that we know how to encode effectively and elegantly grows. Along the way we keep discovering better idioms and abstractions for implementing transformations. Experience shows that external users can succesfully build non-trivial transformation systems using Stratego/XT. With the refactorings and new documentation in Stratego/XT 0.17, it should be accessible and useful to a wider audience.

Software Generation, High-Performance Compilers, and Program Transformation at Utrecht University who suffered imperfect implementation and lack of documentation, provided feedback, and sometimes innovations [16]. About 15 of those students ended up doing a master's thesis project related to Stratego/XT, contributing to the system and research [21,14,11].

References

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

[2] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *3rd IEEE Itl Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 65–74, Amsterdam, The Netherlands, Sep 2003. IEEE Comp. Soc. Press.

[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[4] I. D. Baxter. Design maintenance systems. *Commun. ACM*, 35(4):73–89, 1992.

[5] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[6] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier Science Publishers.

[7] J. M. Boyle and K. W. Dritz. An automated programming system to facilitate the development of quality mathematical software. In *Information Processing 74*, pages 542–546, Amsterdam, 1974. North-Holland.

[8] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser Boston Inc., Cambridge, MA, USA, 1997.

[9] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[10] M. Bravenboer. Connecting XML processing and term rewriting with tree grammars. Master's thesis, Utrecht University, Utrecht, The Netherlands, November 2003.

[11] M. Bravenboer, R. de Groot, and E. Visser. MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In R. Lämmel and J. Saraiva, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *Lecture Notes in Computer Science*, pages 297–311, Braga, Portugal, 2006. Springer Verlag.

[12] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. Components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, Charleston, South Carolina, January 2006. ACM SIGPLAN.

[13] M. Bravenboer, E. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-lr parsing. In W. R. Cook, editor, *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*, pages 209–228, Portland, Oregon, USA, October 2006. ACM Press.

[14] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69:1–56, 2005.

[15] M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In R. Glück and M. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 157–172, Tallinn, Estonia, September 2005. Springer.

[16] M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *LNCS*, pages 237–251, Copenhagen, Denmark, July 2002. Springer.

[17] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proc. the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[18] J. R. Cordy. TXL - A language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, 2004.

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

[20] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.

[21] E. Dolstra and E. Visser. Building interpreters with rewriting strategies. In M. van den Brand and R. Laemmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers.

[22] T. Ekman. Rewritable reference attributed grammars - design, implementation and applications. Licentiate thesis, Lund University, 2004.

[23] B. Fischer and E. Visser. Retrofitting the AutoBayes program synthesis system with concrete object syntax. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 239–253. Spinger-Verlag, 2004.

[24] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: a complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, 1992.

[25] P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.

[26] P. Johann and E. Visser. Strategies for fusing logic and control via local, application-specific transformations. Technical Report UU-CS-2003-050, Institute of Information and Computing Sciences, Utrecht University, February 2003.

[27] M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier, April 2001.

[28] K. T. Kalleberg. www.spoofax.org.

[29] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In N. M.-O. Horatiu Cirstea, editor, *Proceedings of the 6th International Workshop of Rule-Based Programming (RULE)*, ENTCS, Nara, Japan, April 2005. Elsevier.

[30] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[31] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.

[32] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[33] K. Olmos and E. Visser. Turning dynamic typing into static typing by program specialization. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 141–150, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[34] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, volume 3443 of *LNCS*, pages 204–220. Springer-Verlag, April 2005.

[35] T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, 1995.

[36] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[37] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, 2003.

[38] M. G. J. van den Brand, P.-E. Moreau, and J. J. Vinju. Environments for term rewriting engines for free! In *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 424–435, 2003.

[39] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

[40] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag.

[41] R. B. Vermaas. xDoc. An extensible documentation generator. Master's thesis, Utrecht University, Utrecht, The Netherlands, February 2004. INF/SCR-03-41.

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

[43] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[44] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[45] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer, May 2001.

[46] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[47] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.

[48] E. Visser. *The Stratego/XT BibTeX Tools. Tool documentation.* Dept. Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 0.2pre12491 edition, Aug 2005. (Draft).

[49] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[50] E. Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. Tutorial for International Summer School GTTSE 2007.

[51] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[52] E. Visser et al. www.strategoxt.org.

[53] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

[54] D. G. Waddington and B. Yao. High fidelity C++ code transformation. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications*, ENTCS. Elsevier, April 2005.

[55] M. Ward and H. Zedan. MetaWSL and meta-transformations in the FermaT tansformation system. In *COMPSAC 2005: Proceedings of the 29th Annual International Computer Software and Applications Conference COMPSAC 2005*, July 2005.

[56] M. P. Ward and K. H. Bennett. A practical program transformation system for reverse engineering. In *WCRE '93: Proceedings of the 1993 Working Conference on Reverse Engineering,* (Baltimore, Maryland; May 21-23, 1993), pages 212–221. IEEE Computer Society Press (Order Number 3780-02), May 1993.