XT Capita Selecta

Merijn de Jonge   Joost Visser

Abstract

XT is a bundle of program-transformation tools. Stratego is part of this bundle, and is used as implementation language for many tools throughout its packages.

Giving special attention to the role of Stratego, we discuss a selection of our XT experiences. These range from the construction of meta-tools to support Stratego programming, through Stratego techniques applied in constructing some of XT's constituents, to projects where XT (including Stratego and tools programmed in Stratego) has been applied to program-transformation problems.

1 Stratego and XT

Overview of XT XT is a bundle of program-transformation tools [3]. Its purpose is to support component-based development of program transformations. Figure 1 gives an overview of the individual tool packages that are bundled by XT. At the heart of XT

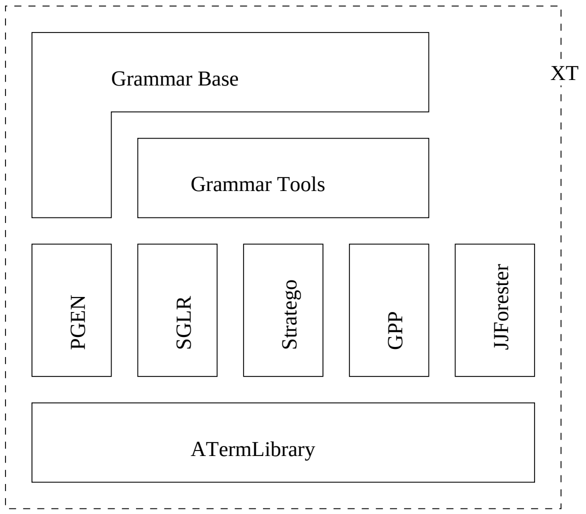

Figure 1: XT is a bundle of packages.

lies the ATERM library [1]. The ATERM format is a generic tree representation format, for which the ATERM library provides space and time efficient support. Within XT, the ATERM format is used as a common exchange format for parse trees and abstract syntax trees.

The Stratego package contains the Stratego compiler [12] and the Stratego standard libraries [13]. ATERMs are used both as input and output of Stratego programs and internally for term representation.

The syntax definition formalism SDF [5, 11] is used throughout XT as grammar formalism. The packages `pgen` and `sglr` contain the parse table generator and generic parser that support SDF. The pretty-print table generator and generic pretty-printer for SDF are provided by the GPP package [2]. All these tools make use of ATERMs as exchange and representation format.

The JJForester package contains a code generator to support representation and traversal of tree structures in Java [9]. Again, the ATERMs are used to represent and exchange trees.

The GrammarTools package contains a suite of tools for grammar analysis, grammar (re)construction, and (parse) tree manipulation. Most of these tools are built from components programmed in Stratego, and instantiations of `sglr` and the generic pretty-printer. Again, ATERMs are used for tree exchange between these components.

The Grammar Base is a collection of syntax definitions in SDF for a growing number of formats, specification languages, and programming languages. The Grammar Base uses `pgen`, `sglr`, and GPP, and various constituents of the GrammarTools.

The philosophy of XT In the design and organization of XT three guiding principles have been followed to realize its purpose of supporting component-based transformation development.

Many, small, stand-alone components The functionality offered by the various packages that XT bundles should as much as possible be available in separate chunks that can be individually (re-)used. Integration of individual pieces of functionality can be left to the user of the package, or can be offered in such a way that separate use of the components is not obstructed.

Simple, common exchange format The input and output of all components should as much as possible be done in a simple, common exchange formats. In XT, the ATERMs are used for this purpose. Having a common exchange format ensures smooth interoperation of components from different packages.

Grammars as contracts The shape of (parse) trees that are represented in the exchange format should be defined in grammars that are independent of specific components, and independent of specific implementation languages. Code for tree representation, for reading and writing trees, for parsing and for pretty-printing should be generated from grammars as much as possible. Thus, grammars should be used as component interface definitions (contracts).

The role of Stratego in XT As the overview of XT already suggests, Stratego fulfills several roles in XT. Firstly, Stratego is a constituent of XT that can be used by transformation developers as a powerful transformation language. Secondly, it is used within XT itself as implementation language for tools in various packages. For instance, almost all grammar tools are implemented in Stratego, as are several components of GPP. Finally, Stratego programming is supported by a number of meta-tools contained in the GrammarTools package.

In the upcoming sections we will shed light on these roles. In Section 2 we explain the Stratego meta-tooling contained in XT. In Section 3 we discuss a selection of the constituents of XT programmed in Stratego. In Section 4 we touch on some programming and engineering techniques related to Stratego that we deployed in the development of XT. Finally, Section 5 evaluates Stratego, in light of its use in XT, and lists some suggestions for further development of the language and its support.

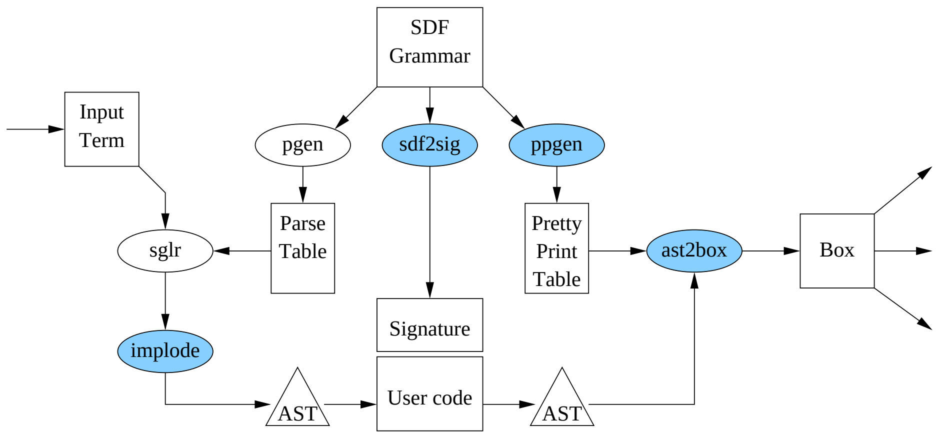

Figure 2: Meta-tools for parsing, signature generation, and pretty-printing integrate Stratego with SDF.

2 Meta-tooling

Apart from the Stratego compiler, a number of meta-tools are available in XT that support Stratego programming. These include tools that integrate Stratego with SDF, and tools that analyze the import structure of Stratego programs.

Integration with SDF Figure 2 shows an overview of the meta-tools offered by XT that integrate Stratego and SDF. In combination, these tools instantiate a meta-tooling architecture where grammars are used as contracts between components [4]. The shaded ellipses are tools that are themselves programmed in Stratego.

Normally, the Stratego programmer manually constructs the signatures he needs. Also, the input and output of his programs are ATERMS. With the meta-tooling offered by XT, the programmer can construct front and back-ends to his programs for parsing and pretty-printing, and he can generate the signatures he needs from the grammars he uses. In general, the programmer follows the following steps:

- The programmer constructs or reuses an SDF grammar. From this grammar, he generates a parse table, a signature, and a pretty-print table, using the tools `pgen`, `sdf2sig`, and `ppgen`.
- The programmer imports the generated signature into his program.
- A concrete input term is fed into the generic parser `sglr`, together with the generated parse table. The resulting parse tree is imploded to an abstract syntax tree (AST) using the `implode-asfix` tool. As this AST is represented by an aterm, it can be consumed by the Stratego program.
- The output of the Stratego program is pretty-printed in two steps. First, the AST is transformed into a formatted BOX expression by `ast2box`, using the formatting rules in the generated pretty-print tables (possibly supplemented with customizations of the user). Secondly, the box expression is transformed to plain text, HTML, or LaTeX. This is done with the BOX back ends, which are not pictured.

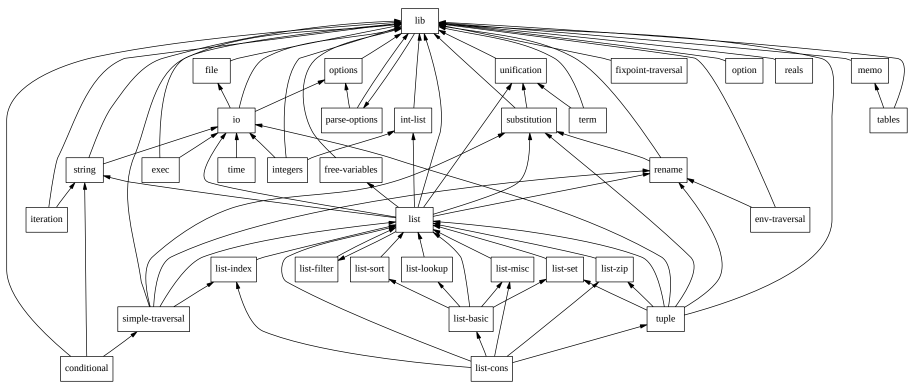

Figure 3: Generated import graph for the Stratego library.

Note that the parse and pretty-print tables are not Stratego-specific. Consequently, the parsing and pretty-print back-ends can be used for any component that consumes and produces aterms. Analogous to `sdf2sig` for Stratego, SDF-driven code generators exist for other programming languages, e.g. JJForester for Java. Stratego programs can seamlessly interoperate with programs in those languages, because they use code generated from the same grammar and employ the ATERM format as exchange format.

Import analysis Using Stratego's import mechanism, Stratego programs can be built from several modules. For various purposes, the import relations that exist among modules need to be analyzed. One of these purposes is compilation. In fact, one of the earliest phases of the Stratego compiler is *packing*, i.e. collecting all imported modules into a single file. The `pack-stratego` component, which takes care of packing, is actually a Stratego-specific instantiation of the generic meta-tool `pack`, which is part of XT. Another instantiation is `pack-sdf`, which collects SDF modules into a single SDF definition file. Apart from a file with packed modules, the `pack` tool also produces a dependency file, which can be included in a Makefile.

Another purpose for which import analysis is useful is program analysis and program comprehension. For these purposes, XT offers another generic tool, called `import`. When instantiated for Stratego, it analyses the import relations for a given program, and presents the analysis results as a list of module names, a list of full file names, or an import dependency graph. For instance, the generated import graph for the Stratego library `lib` is show in Figure 3. Note that this picture immediately reveales two mutual imports (of `lib` and `parse-options`, and of `list` and `list-filter`) that are probably unintended.

Both `pack` and `import` make use of the generic graph strategies that are contained in the Stratego library. These two tools have much functionality in common. We consider combining them in a single tool that is not only generic with respect to the input language, but also with respect to the actions performed after analysis: result presentation or packing (several dimensions of genericity).

3 XT constituents

Apart from the above-mentioned meta-tools for Stratego, XT contains a wide variety of transformation components that have been programmed in Stratego. We will explain a selection of these components for graph transformation, tree transformation, and grammar transformation.

3.1 Graph transformation

Graph formats Graph formats and corresponding tooling play an important role within XT for code analysis and code visualization. According to the component based idea, XT reuses existing graph visualization tooling based on the `dot` input format [8]. The `dot` format is a low-level graph representation language, which contains many details concerning the visual appearance of graphs. To simplify tool construction, XT offers an alternative, high-level graph format GRAPHXML [6], in which a graph can be defined in terms of its mathematical description only (i.e. in terms of nodes and edges). XT also offers the necessary conversion tooling to transform a graph represented in GRAPHXML to `dot`. For both formats SDF grammars are available in the Grammar Base and with XT's meta-tools mentioned above, parsers, pretty-printers and Stratego signatures have been generated. These have been used to construct graph transformation components.

Converters To convert between graph representation formats, several graph transformations have been programmed, including `GraphXML2dot`.

Analysers The Stratego module `GraphXML-analysis` contains a number of reusable strategies that perform analysis on graphs represented in GRAPHXML. The transformation component by the same name performs a number of these analysis strategies, and presents the results as plain text. Most of these analyzers are implemented using the standard set strategies as contained in the Stratego library in combination with `collect`. For example, the strategy to obtain the source nodes of a graph, was implemented as follows:

```
get-sources = collect( \ source(x) -> <de-quote>x \ )
```

Similarly, target nodes can be obtained using the `get-targets` strategy, implemented as:

```
get-targets = collect( \ target(x) -> <de-quote>x \ )
```

Both strategies can be combined to form more advanced analyzers. To obtain those nodes from a graph with only in-going edges (sinks), the `get-sinks` strategy was implemented:

```
get-sinks
  = \g -> <diff>(targets,sources)
     where <get-targets>g => targets
         ; <get-sources>g => sources
    \
```

Extractors A number of components are available in XT for extracting graphs from source code. An example is `sdf2sg`, which extracts a sort dependency graph (or sort graph) from an SDF grammar. The extracted graph is represented in GRAPHXML. The utility `get-sort-graph` is a script that glues the extractor `sdf2sg` with the converter `GraphXML2dot` and the graph formatter `dot`.

3.2 Tree transformation

Tree visualization The component `treeviz` takes an arbitrary ATERM, and produces a visualization of it in the GRAPHXML format. This is accomplished by decomposing each term $f(a_1, \ldots a_n)$ using the '#' construct of Stratego and creating edges $f \rightarrow a_i$ for each argument ter $a_1 \ldots a_n$:

```
NodeToEdges =
   ?f#( args );
   !args;
   map(NodeToEdge(!target(<quote>f)))

NodeToEdge(target) =
   ?f#(_);
   !edge([source(<quote>f), <target>()] )
```

Parse tree implosion The component `implode-asfix` takes a parse tree in the AsFix format, and produces a slimmed down syntax tree. Various command line options allow control over which implosion filters are activated. When all filters are active, the parse tree is transformed into an abstract syntax tree, which contains no layout or literals, and in which lexical parse trees have been flattened to strings.

The `implode-asfix` component demonstrates the use of command line options in Stratego and the sequential application of filters to an input term. To define command line options in addition to the standard set of options, we use the strategy `iowrapO` and pass it all additional options. The standard Stratego options do not need to be passed to `iowrapO`.

```
main =
   iowrapO( implode, Option( "--lex",    !FlatLex )
                   + Option( "--layout", !RemoveLayout )
                   + Option( "--lit",    !RemoveLit )
                   + Option( "--appl",   !ReplaceAppl )
                   + Option( "--inj",    !FlatInj )
                   + Option( "--list",   !FlatList )
                   + Option( "--seq",    !RemoveSeq )
                   + Option( "--pt",     !RemovePT ) )
```

The `iowrapO` strategy passes a tuple to its argument strategy (`implode` in the example) consisting of the list of command line options as specified by the user and the input term. The strategy should also return a tuple with the arguments and the transformed term. The strategy `implode` accesses the command-line options to determine which filter to apply to the input term. The strategy `option-defined` is used to check whether an option was specified or not. On success, a particular filter is applied, on failure the next option is checked.

```
implode =
   ?term;
   try((option-defined(FlatLex),      flat-lex));
   try((option-defined(RemoveLayout), remove-layout));
   try((option-defined(RemoveLit),    remove-lit));
   try((option-defined(FlatList),     flat-list));
   try((option-defined(ReplaceAppl),  replace-appl));
   try((option-defined(FlatInj),      flat-injections));
   try((option-defined(RemoveSeq),    remove-seq));
   try((option-defined(RemovePT),     remove-pt));
   try(?term; (id, implode-asfix))
```

When no options were specified, the input term remains unchanged after all option checks have been performed. The last match in the example, which tries to match the same input term as the first match therefore succeeds, and a default implosion will be applied.

Syntax tree formatting To format parse trees and abstract syntax trees, BOX front-ends `asfix2box` and `ast2box` are available. Apart from a tree, these utilities take a sequence of pretty-print tables as input. The output is a term in BOX, a language independent markup language to describe the intended layout of text. A BOX term can be passed to one of the BOX back-ends to obtain HTML, LaTeX or plain text as output.

The BOX front-ends use Stratego's table mechanism for efficient storage and retrieval of pretty-print rules. During initialization, a global accessible table is created with `create-table` and filled with all pretty-print rules defined in the pretty-print tables:

```
read-pp-tables =
   ?names;
   <create-table>"pp-table";
   <map(ReadFromFile; build-pp-table)>names
```

Pretty-print rules are inserted in the table with the `table-put` strategy:

```
StoreEntry =
   ?PP-Entry( path, template );
   <table-put>("pp-table", <mk-key>path, (path, template))
```

After initialization, the pretty-print rules are accessed with `table-get` to transform an abstract syntax tree or parse-tree to BOX:

```
pp-table-get =
   ?key;
   <table-get>("pp-table", key ) => (path, template)
```

3.3 Grammar transformation

Grammars play an essential role in XT. They are used to generate parsers, pretty-printers, Stratego signatures, and Java libraries for language-specific tree transformation. In fact, grammars are used to fix the interfaces between transformation components [4]. To support this employment of grammars, we need tools to create, manipulate, and transform grammars, and to generate code from grammars. We discuss a few of these tools.

Syntax definition In XT, we use SDF as primary syntax definition formalism. Because of its purity and declarativeness, grammars written in this formalism are well suited to be used for different purposes (parsing, pretty-printing, code generation), as is desired in XT. Also, its modularity and expressiveness enables syntax reuse.

Grammar extraction Grammars can of course be written manually from scratch, but when a language definition in a different formalism is already available, an SDF definition can be generated in stead. The GrammarTools include several components for this purpose. The tool `yacc2sdf` extracts an SDF grammar from a Yacc parser description. The tool `happy2sdf` does the same for the Yacc-derivative Happy that targets Haskell. This latter tool is actually a composition of `happy2yacc` and `yacc2sdf`.

In XML, document type definitions (DTDs) are used to describe the structure of XML documents, i.e. to describe the abstract syntax to which they must conform. The component `dtd2sdf` extracts an SDF definition from such a DTD. From the extracted grammar, a validating[1] parser can be generated to parse XML documents that conform to the given DTD. From the same grammar, a DTD-specific Stratego signature can be generated, to support development of DTD-aware Stratego components.

Grammar rephrasing Rephrasings are transformations where the source and target language coincide, or largely overlap. Example of a grammar rephrasings are the tools `sdf2asdf` and `sdf-normalize`, which both map SDF to a subset of SDF.

Other examples of rephrasings are `sdf-cons` and `sdf-label`, which synthesise labels and constructor names from SDF productions, and add these to a grammar, and the converse tools `rm-cons` and `rm-labels`. The latter tool makes use of Stratego's overlay construct to simplify programming on parse trees in the verbose parse tree format AsFix. The grammar of the SDF language contains the following production for labelled symbols:

```
  Literal ":" Symbol -> Symbol {cons("label")}
```

Using the meta-tool `sdf2overlay`, which is an enhanced version of `sdf2sig`, the following Stratego code can be generated:

```
signature
  constructors
    label : Literal * Symbol -> Symbol
overlays
  label-overlay(u_20,v_20,w_20,x_20)
    = appl(prod([cf(sort("Literal")),cf(opt(layout)),
      lit(":"),cf(opt(layout)),cf(sort("Symbol"))],cf(
      sort("Symbol")),attrs([cons("label"),'id(
      "Label-Sdf-Syntax")])), [u_20,v_20,appl(prod([
      char-class([58])],lit(":"),no-attrs),[58]),w_20,
      x_20])
```

The right-hand side of the generated overlay is the AsFix fragment that represents the concrete parse tree that represents labeled symbols. Given this generated overlay, the tool that removes labels is now simply programmed as follows:

```
  rm-labels
    = topdown( try(\label-overlay(_,_,_,d) -> d\ ) )
```

Thus, the overlay is used to match labeled symbols, select their subterm that represents the symbol without label, and replace them with this subterm.

4 Programming and engineering techniques

During the development of XT, we employed a range of programming and engineering techniques that are more or less specific to Stratego. Though these techniques are hardly novel, and certainly not earth-shaking, they might be of interest to other (potential) Stratego practitioners. For that reason, we discuss them.

[1]An XML parser that enforces a document's conformance to a DTD is called a *validating* parser.

Learning the ropes To learn Stratego, a wide range of sources is available. In our experience, the best starting place is the Stratego Tutorial [14], in combination with the Stratego Library [13]. To go beyond first principles, it is best to consult examples of systems programmed in Stratego, such as the Stratego Compiler itself [12]. We can also recommend the GrammarTools package of XT, as it contains a large number of fairly small components. The Stratego homepage is implemented as a Wiki server. There you can find (and add) information that is complementary to the official sources, and discussions about Stratego practice. If you really get stuck, you can post an SOS message on the Stratego mailing list.

Understanding and reusing code Reuse is better than programming from scratch. However, to reuse strategies, one first needs to understand them. In our experience, this can be quite difficult. To find out, for instance, how to invoke a strategy, one needs to know what kind of terms it expects, and what kinds of strategies are needed to instantiate its parameters. Lacking a type system or a documentation standard, such information is difficult to glean from the code.

To reconstruct such 'usage information', two complementary tactics can be followed: look at the definition of the strategy, or look at its call sites. In our experience, the second tactic is easier and more often successful. To find call sites of strategies, one can perform searches in code bases, or PDF documents.

Despite the lack of a proper mechanism for code understanding in Stratego, developing reusable code is simplified thanks to the modularization mechanism, argument strategies, and untypedness. Stratego's standard library provides a great number of reusable strategies which significantly decreases the amount of code that needs to be written for each application. Figure 3 depicts the import graph of the Stratego library and gives an impression of the diversity of he functionality of this library.

Choosing the proper construct Often, different language constructs can be chosen to express the same. Which construct to use depends on personal taste, there usually is no general rule to follow. Below we discuss a few of such constructs.

- A Stratego rule is syntactic sugar for a strategy which starts with a match and ends with a build construct:

  `L:l->r where s` is *equivalent* to `L={x1,..,xn: ?l;where(s);!r}`

  A Stratego rule looks similar to a rewrite rule, which transforms a term from left to right. This similarity can be used as heuristic to choose between a rule and a strategy: use a rule when a transformation is similar to a rewrite step.

- Arguments to rules and strategies can be passed through strategy variables or by tupling. For instance, the `iowrap0` discussed before uses two strategy variables, whereas the strategy `implode` that is called by `iowrap0` is passed a tuple with the specified command line options and the input term. Tupling arguments usually requires explicit matching and construction of the tuple as in the example below:

  ```
  ?aterm;
  <strategy_a>(arg1, ..., argi, aterm);
  ?result;
  <strategy_b>(argj, ..., argn, result)
  ```

With strategy variables, the two matches are not required, and the code looks more like the sequential application of transformations:

```
strategy_a(arg1, ..., argi);
strategy_b(argj, ..., argn)
```

- To construct and de-construct terms in Stratego the match (‘?’) and build (‘!’) operators can be used. For example, to construct a term t’ using a sub-term s occurring in a term t, the following Stratego construct can be used:

  ```
  ?t(..., s, ... );
  !t’( ..., s, ... );
  ```

  This can also be done using the anonymous rule construct:

  ```
  \ t(..., s, ...) -> t’( ..., s, ...) \
  ```

  This similarity allows rule-like definitions as strategies. See the definition of `get-sinks` for an example. It uses an anonymous rule in a strategy definition for matching the input term and building the output term.

- When the same term should be used at several places in a strategy definition, the term needs to be preserved before its first use until its last use. This can be achieved by assigning the term to a Stratego variable:

  ```
  ?new_term
  s_1;...;s_n
  !new_term
  ```

  In this example, the input term is saved before executing the strategies $s_1 \ldots s_n$ and restored afterwards. The same can be achieved without introducing additional variables using the `where` construct. For example, to search for a pattern in a term and assigning it to a variable `pattern` without affecting the input term, the following Stratego code can be used:

  ```
  where( oncetd(?f(a,b)=>pattern))
  ```

Debugging heuristics Debug support in Stratego is rather minimal, and consequently errors are usually hard to locate. A few simple techniques are available to help debugging Stratego programs, but debugging remains a time consuming business.

- Stratego offers the `debug` strategy, which writes the subject term to standard error. This strategy accepts a string as optional argument which is written before the subject term and helps to distinguish between different `debug` invocations. An often occurring mistake is to forget to build the argument string (by omitting the Stratego build operator). In this case no debug output is displayed and you might conclude incorrectly that the debug strategy is not reached because of a bug somewhere before. It is because of the incorrect use of `debug` however that no debug output is displayed, not necessarily due to a bug in your program.

- Often you believe that a subject term matches a particular pattern but somehow a strategy operating on that term fails. Is the strategy incorrect, or does the input term not match the expected pattern? An input term can be displayed with the `debug` strategy, but checking the output of the `debug` strategy by hand is error prone. One can also pass an explicitly built input term of the proper structure to the strategy and check whether the strategy still fails or not. Similarly, one can check the result of a strategy by inspecting the output of the `debug` strategy, or by explicit matching:

  ```
  // Explicitly build a term to check a strategy:
  !my_term(arg_1,...,arg_n);
  s;
  // Explicit match to verify that the structure of
  // the output term of s is as expected.
  ?a_term(arg_1,...,arg_j)
  ```

Unit, component, and integration testing Testing of Stratego programs can be performed using unit, component, and integration tests.

- The Stratego library provides *unit testing* using the `test-suite` strategy contained in the library module `sunit`. This strategy performs a number of tests and produces a test report as output. The `sunit` module contains standard tests for testing that a strategy succeeds for a particular input or that it fails, and for checking the result of a strategy.

  ```
  main = test-suite( !"my-unit-tests",
                     test-1;test-2;test-3)

  test-1 = apply-test(!"test1", id, !"some input")
  test-2 = apply-and-fail( !"test2", fail, !"some input")
  test-3 = apply-and-check( !"test3", sqrt, !4.0, ?2.0)
  ```

  Such test suites can be defined in separate modules as is the case for part of the Stratego library where a module `m-test` defines a test suite for the strategies in the module `m`. One can also define the test suite in the same module and turn it on using the `--main` switch of the Stratego compiler.

  For example, consider the module defining the strategy `some-strategy` and a corresponding test suite below:

  ```
  main = iowrap(some-strategy)

  test-some-strategy = test-suite( .... )
  ```

  Testing of `some-strategy` can be turned on by re-compiling the program with the `--main` switch:

  ```
  sc -i program.r --main test-some-strategy
  ```

- To test a complete Stratego program (*component testing*) it can be passed several fixed input terms, and the output can be matched against corresponding pre-build and verified result terms. In XT we use the test mechanism of Automake [10] which executes a sequence of programs implementing tests and builds a status report. We use `diff` to compare the output of Stratego programs with pre-build terms.
- Most Stratego programs in XT operate on parse-trees or abstract syntax trees and the functionality of these programs is usually small because of the component based approach of XT. Complete programs can be constructed by connecting parsers and pretty-printers to Stratego components, and by combining multiple (Stratego) components together. To test such complex compositions of components, XT defines a collection of *integration tests*. These tests execute the components of XT in different combinations with different inputs and monitor the consistency of XT on a daily basis.

Refactoring When a piece of code from one program is candidate for reuse in another program, it is extracted and put into a separate module. This module is put in a lib directory of GT (GrammarTools). Modules in GT/lib expected to be useful outside GT, can be nominated as candidates for the Stratego library. This way, GT functions as a kind of nursery for the Stratego library.

Adding command line options Command line options and switches are handled automatically when using one of the `iowrap` strategies. By using one of these, the following options/switches are accepted and handled by default:

-b			Write output in binary (BAF) format
-h	-?	–help	Display usage information
-i `<file>`		—input `<file>`	Read input term from `<file>`
-o `<file>`		—output `<file>`	Write output term to `<file>`
-s			Write statistic information after execution
-S		—Silent	Silent execution
-v		—version	Display version information

In addition to these standard command line options, new options can be defined. The `iowrapO` strategy provides a simple mechanism to specify additional switches. Section 3.2 contains an example that demonstrates the use of the `iowrapO` strategy and the definition of extra switches. Unfortunately, the generic option handling mechanism of Stratego currently does not support extension of the usage information. Hence, even when new options are accepted, the `--help` switch only displays usage information for the standard set of options.

Optimization The use of standard traversal strategies usually makes programs short and limits language dependence. However, it is often difficult to choose between all different traversals that are provided by Stratego. When several candidate traversal strategies are available to perform a particular job, choosing one or another can influence performance significantly due to their specific traversal pattern.

To control the exact behavior of a traversal an obvious approach is to explicitly program it. This is of course somewhat against the Stratego way of thinking, but when performance really matters it might be an approach to follow.

An alternative approach is to use the *skip* strategies. Ordinary traversal strategies do not distinguish between different nodes and during traversal all nodes are accessed. Usually, a transformation only operates on particular sub trees and only a traversal of these trees would suffice. The *skip* strategies accept an additional argument which determines the trees that should be traversed.

For example, the `asfix-yield` program obtains the yield of an AsFix parse tree which corresponds to the original input string. AsFix is a huge parse tree format and most of the information it contains is not used for this transformation. By traversing only those parts of the tree that contain lexical information, performance is improved significantly. Therefore, `asfix-yield` is implemented by defining a skip strategy for the AsFix constructs that only need partial traversal and we define which sub-trees of these constructs should be traversed:

```
asfix-yield1 =
      leaves(printstring, is-string, skip1)
skip1(x) =
      term(id,id,id,id,id,x,x,x,id)
      + appl(id, id, list(x))
      + list(id, id, list(x))
      + lex(x, id)
```

This `skip1` strategy defines that only three of nine sub-terms of a `term` constructs need to be traversed. Similarly, of an `appl` construct only one of three sub-trees will be traversed.

Connecting components Individual XT components should be designed for reuse according to the component based approach of XT. Consequently, their functionality usually will be restricted. Advanced programs can be constructing be connecting such reusable components together using the ATERMS as exchange format.

Several techniques are used in XT for gluing XT components together. Unix pipes and scripting are used most frequently. Unix scripts handle user interaction (including option handling) and execute all components. Pipes are used to transfer data between components. Component gluing with `make` and Makefiles is another frequently used technique. The individual steps involved in the execution of programs can be expressed clearly in Makefiles as make dependencies. This improves understandability of programs in comparison to Unix scripts. In contrast to Unix pipes, `make` uses files to pass data between components. The combination of dependency checking and data exchange by files enables automatic program optimization by only executing components when a re-computation is required. In contrast to Unix scripting, option handling with `make` is difficult.

A third approach being used in XT is Stratego as scripting language to glue components together. This is a somewhat experimental approach because Stratego does not offer real powerful process strategies yet. It offers the `call` strategy to create a new process and execute an external program, data passing between programs has to be programmed explicitly.

For example, GT defines a strategy `sglr` which uses `call` to execute the parser `sglr`:

```
sglr : (tbl, in, out) -> out
       where
         <call> ("sglr", ["-p", tbl, "-i", in, "-o", out])
```

The terms `in` and `out` denote file names of the input and output terms. A Stratego program is responsible for passing data to `sglr` using files. Execution with input/output redirection is not supported.

The option handling mechanism of Stratego and the ability to modify intermediate trees between execution of components are strong motivations for using Stratego for component gluing.

To simplify tool construction and tool use, XT requires that all components performing input/output accept the `-i`, and `-o` switches to specify input and output, respectively. Furthermore, tools should also handle the `-h` to provide brief usage information to the user.

Documentation General documentation about XT, component descriptions including usage information, and tasks descriptions are created and maintained using Wiki [2], a system for collaborative web development. Wiki makes documentation writing and updating extremely simple and there is no delay between documentation writing and documentation availability.

Stratego offers literate programming [7] facilities which can be used to document Stratego programs. Together with a LaTeX style file, this provides the ability to produce LaTeX documents from your Stratego programs. A literate Stratego program starts with `\literate[<module-name>]`, which indicates the start of an ordinary LaTeX file. Code should be placed between `\begin{code}` and `\end{code}`.

```
\literate{my-stratego}
Here comes a program description in \LaTeX.

\begin{code}
strategies
   main = iowrap(id)
\end{code}

The \LaTeX document continuous here
```

When you include literate programs in you LaTeX document, you should add `\usepackage{lit-style}` to the document preamble. The Stratego parser ignores all text except for the text between `\begin{code}` and `\end{code}`. An increasing number of XT components are developed as literate programs.

Usage information about individual components can be obtained with the `-h` command line switch. Specifying the `-h` switch provides a brief listing of available switches. For most components however, also a separate Wiki page is available, which contains additional usage information. To obtain information about or related to a tool, go to our XT web-site at `http://www.program-transformation.org/xt` and enter the tool name in the search entry at the bottom of the Wiki page. This gives you a list of all Wiki pages where the tool name is used and this is an easy way to find the desired information.

5 Conclusions

Given our experiences with Stratego in the context of XT, we will attempt to evaluate the language's strengths and weaknesses. Also, we will suggest some possible

[2] `http://c2.com/cgi/wiki`

improvements.

5.1 Strengths and weaknesses

Strengths Stratego has proven to be a very powerful implementation language within the context of XT. This is due to various strong features of Stratego.

- *Genericity*. Strategy parameters and Stratego's untypedness allow the development of generic reusable strategies. The standard library of Stratego best demonstrates how this helps to shorten your programs and it demonstrates the development of such strategies.
- *Language independence*. With the generic traversals offered by Stratego your programs only need to depend on those language constructs that require transformation. This simplifies the development of transformations significantly when only a relative small part of a language is affected. It also simplifies maintenance of Stratego code after a language change.
- *Large libraries*. The standard libraries in Stratego's distribution offer an extensive set of reusable generic strategies. Though not all of these strategies are reused as often and with the same ease, they significanlty reduce programming effort.
- *Common exchange format* Stratego supports the ATERMS as exchange format. This is in line with the philosophy of XT, and greatly simplifies component-based development.

Weaknesses We also identified a number of weaknesses of Stratego, and the support that is currently available for it.

- *Performance*. One of the most problematic properties of Stratego is the poor performance of its compiler. This has great impact on the development effort of Stratego programs because the development cycle editing–compiling–testing takes so much time.
- *Static checking*. Due to its untypedness, static checking of Stratego programs is limited and there is no way to see what type of data is expected by strategies. This often results in unexpected behavior of Stratego programs.
- *Conceptual complexity*. Stratego is a difficult language to learn because programming with strategies is a technique that most people are unfamiliar with, because the syntax of the language is uncommon, language documentation is still under development, and because of minimal debugging facilities and poor error reporting. Consequently, it takes quite some time and effort to get used to the language.
- *Code comprehension and reuse*. Lack of a type system and minimal documentation of library strategies hampers reuse. Although there is a huge amount of strategies available for reuse, it is hard to find out whether a strategy with a specific functionality is available and how to use it. Finding out what input a strategy expects and what output it returns are re-occurring problems.

5.2 Suggestions for improvement

- *Additional checks and error reports*. Short of a type system for Stratego, and an accompanying type-checker, many additional static checks would be helpful to the programmer. For instance, mistakes in arities of strategies and strategy variables could be caught and reported as such by the compiler, instead of leading to C compilation or linking errors.

- *Number of rewrites*. Although the traversal strategy that is used can have great impact on the performance of Stratego programs, there is no mechanism (except for time measurements) to compare the use of different strategies. Additional statistics, like statistics about the number of rewrites for instance, would be of great help.

- *Globally accessible options*. Code depending on command line switches as specified by the user can occur everywhere in a program. The option handling mechanism of Stratego now requires that the list of options is passed through the program. This mechanism pollutes Stratego programs and may affect many strategy definitions. Making the options globally available would simplify Stratego programs, because it makes accessing the command line switches easier and eliminates the need to pass switches through a program.

- *Documentation*. Reuse of strategies contained in the the Stratego library is hampered because it is often not clear how to use a strategy. Adding examples of strategy usage would improve the reusability of the library. Also, a documentation standard for Stratego strategies is required to indicate the arities of strategy argument, types of input and output terms, and the types of variables. A *reference card* listing the most commonly used strategies, and language constructs would assist the novice Stratego user in learning the language.

References

[1] M. van den Brand, H. Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[2] 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.

[3] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In *Language Descriptions, Tools and Applications (LDTA 2001)*. Electronic Notes in Theoretical Computer Science, To appear.

[4] M. de Jonge and J. Visser. Grammars as contracts. In *Generative and Component-based Software Engineering (GCSE)*, Lecture Notes in Computer Science (LNCS), Erfurt, Germany, To appear. Springer.

[5] 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.

[6] M. M. I. Herman. GraphXML – An XML-based graph description format. In *Symposium on Graph Drawing (GD 2000)*. Springer, 2000. To appear.

[7] D. E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[8] E. Koutsofios. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, Nov. 1996. This report, and the program, is included in the `graphviz` package, available for non-commercial use at `http://www.research.att.com/sw/tools/graphviz/`.

[9] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Language Descriptions, Tools and Applications (LDTA 2001)*. Electronic Notes in Theoretical Computer Science, To appear.

[10] D. Mackenzie and T. Tromey. Automake. Available from `http://www.gnu.org/`.

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

[12] E. Visser. The Stratego compiler. Technical report, Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.

[13] E. Visser. *The Stratego Library*. Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.

[14] E. Visser. *The Stratego Tutorial*. Department of Computer Science, Universiteit Utrecht, Utrecht, The Netherlands, 1999.