TETE: A Non-Invasive Unit Testing Framework for Source Transformation*

Derek M. Shimozawa James R. Cordy
School of Computing, Queen's University, Kingston, Canada
E-mail: dshimoza@ca.ibm.com, cordy@cs.queensu.ca

Abstract

While the use of test-driven development as a debugging, pedagogic, and analytical methodology for object-oriented and procedural systems is well documented, it is a relatively unexplored and informal practice within the paradigm of source transformation. This paper describes a test-driven approach to the specification and evaluation of source transformation programs through rule-by-rule and type-by-type unit testing. We introduce the Transformation Engineering Toolkit for Eclipse (TETE), a test-driven framework centered around a simple yet flexible infrastructure for automatically and non-invasively unit testing subtransformations, application strategies, and grammar types specified in the TXL source transformation language.

1 Introduction

Test-driven development is an approach to coding which encourages developers to write one or more unit tests to capture and validate the behavior of a program in small, manageable chunks before adding new code. While it has proven well-suited to development of object-oriented and procedural systems [8, 18, 19, 7], it has yet to be explored in the realm of source transformation programming.

Many common software development tasks can be characterized as source transformations, from simple text pretty-printing to advanced source code processing and compilation. The source transformation programming paradigm models these processes as syntactic conversions from the problem domain, represented as program input, to the solution domain, represented as program output, using syntax-based pattern matching and replacement.

In this paper, we describe a test-driven approach to source transformation which leverages unit testing to help overcome some of the existing complexities and learning barriers inherent in many transformation-based systems. We have identified five challenges specific to source transformation that are well suited to unit-testing solutions:

Grammatical type evaluation: Transformation systems use a context-free grammar to tokenize and parse the source input. For generality, their grammar frameworks usually allow for unrestricted grammatical forms, without analyzing or checking for type ambiguities or parser restrictions. Due to this leniency, it is easy for developers to introduce unexpected parsing behavior. This problem is further complicated by the fact that nonterminal types cannot be directly accessed independently of the rest of the grammar and the rules that use it. In the context of a transformation, visibility of a grammatical type is ultimately determined by the application scope of the rules that apply it, and so considerable effort must be expended to test the correctness of a nonterminal type definition as it relates to concrete examples of input.

Rewrite subtransformation evaluation: Transformation processes are implemented using sets of individual subtransformations (rewriting rules) that map syntactic substructures in the program input to those in the output. These rules carry out the bulk of the transformation work on intermediate tree structures (i.e., parse trees) by searching for subtrees matching a specified pattern and substituting them with replacement subtrees constructed from parts captured in the pattern. By default, substitutions are applied in an aggregate fashion to form the final transformed tree structure, and thus the behavior of a single pattern-replacement pair cannot be easily dissociated from that of the overall transformation. When a rule is failing to match its intended pattern, it takes considerable hand effort to localize and debug the offending subtransformation.

Application strategy evaluation: Controlling the application of rules within a transformation is a major concern for programmers. Rewrite systems are not necessarily confluent or terminating, and therefore they often include traversal control facilities and application conditions to constrain the order and scope of application of rewrites. Tracking down errors in a program's application strategy is time consuming, because the entire transformation must be debugged as a whole, as opposed to isolating constituent parts

*This work was supported by the Natural Sciences and Engineering Research Council of Canada and by IBM through an Eclipse Innovation Award

of the overall rule application. In many cases, programmers will attempt to fix their application strategies without fully understanding the precise nature of the error. The result is repeated shotgun re-arrangement of the order of rewrites and re-execution of the entire transformation until it yields the desired results.

Grammar scope refinement: Grammars for common programming languages consist of scores of non-terminal types. Typically, programmers are only interested in a small subset of the grammar's types, because subtransformations operate on only part of the intermediate tree structure at a time. Thus test inputs are often much more elaborate and complex than they need to be when a user is analyzing or debugging the effects of a single rewrite.

Transformation pedagogy: Source transformation systems often help experienced developers to quickly produce source transformations. However, they typically presume a strong grasp of the underlying paradigm fundamentals and place their target audience well above the introductory level, making them difficult to learn. Because of their close conceptual proximity to general compiler technology, they necessitate a radical cognitive shift from procedural and object-oriented programming. Allowing for piecewise transformation through unit testing can be an effective pedagogical solution to help new users compose example tests and answer questions about the evaluation semantics of the source transformation paradigm.

To address these issues, we have designed the Transformation Engineering Toolkit for Eclipse (TETE), a set of Eclipse [11] plug-ins that provides a flexible but simple infrastructure for automatically and non-invasively unit testing rewrite subtransformations, rewrite strategies (which are implicitly defined by the composition of subtransformations), and grammar types written in the TXL source transformation language.

2 The TXL Source Transformation Language

TXL [4, 5, 6] is a special-purpose language designed to provide rule-based source transformation using functional specification and interpretation. A TXL program has two main parts: a context-free grammar that describes the syntactic structure of inputs to be transformed, and a set of context-sensitive, example-like transformation rules organized in functional programming style.

The TXL processor (Figure 1) is a compiler and run-time system that directly interprets TXL programs. Source processing is performed in three phases. First, the parser takes the entire input, tokenizes it, and parses it according to the TXL program's grammar definitions to produce an intermediate parse tree. The second phase transforms the parse tree into a new tree that corresponds to the desired output.

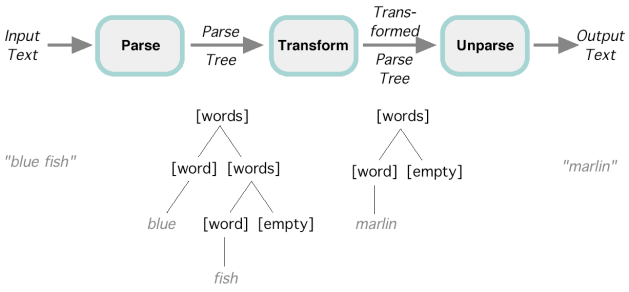

Figure 1. The TXL transformation process.

Finally, the processor unparses the intermediate parse into output text.

3 General Approach

The general approach to unit testing with TETE operates in two phases. The first of these applies TXL's functional decomposition to isolate rule and grammar constructs of interest. The second phase uses Eclipse's [11] source modelling and interface framework to automate the program mutation process and implement each testing feature. Based on a source model for TXL transformations, TETE's unit testing framework calculates the file inclusion dependencies and implements appropriate program mutations to support each test. The details of this approach are explained in the following sections.

3.1 TXL Rule Isolation

Program transformation in TXL is specified using a set of transformation rules that by default use the fixed point compositional semantics of pure rewriting systems. A rule in TXL looks like this:

```
rule name
   replace [type]
      pattern
   by
      replacement
end rule
```

A rule searches its scope of application (the tree to which it is applied) for nodes of the type of the rule. Each time a node of the rule's type is found, the tree rooted at that node is compared against the rule's pattern. If it matches, then the rule builds a replacement tree, and substitutes this for the subtree that was matched, yielding a new scope.

Structurally, TXL rules are organized into a rooted pure functional program in which lower level rules are applied as functions of subscopes captured by higher level patterns [6].

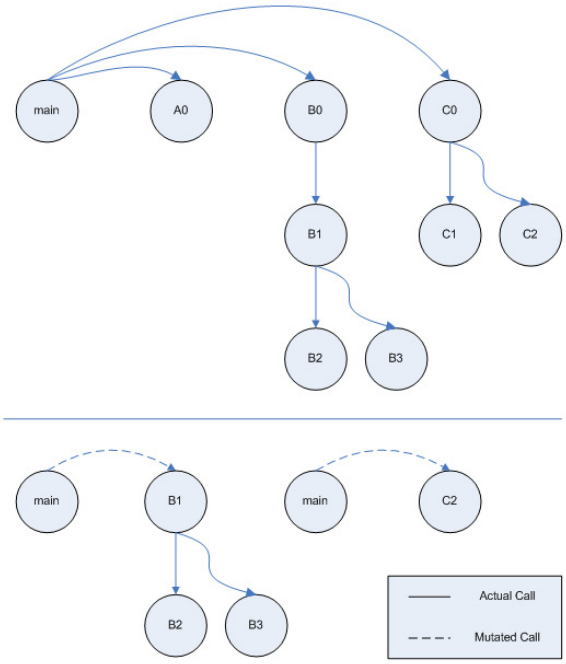

Figure 2. Isolating TXL rules through functional decomposition.

Several rules can be applied to the same scope in succession using functional composition, for example:

$$X[f][g][h]$$

Traversal strategies are implicitly programmed as part of the functional decomposition of the transformation rule set, which controls how and in which order subrules are applied. That is, f is applied first, then g to the result of f, then h to the result of g. In more conventional function notation, the composition would be written as:

$$h(g(f(X)))$$

Our goal is to isolate rewrite rules to be tested so that they can be applied independently of their higher level rules. A rule that is to be unit tested is called the *target rule*. Our hope is that since TXL uses functional decomposition to specify the application and ordering strategy for the program, we can break down the program's application strategy into standalone regions (rooted at the respective target rules) that can be evaluated separately.

Figure 2 illustrates our unit testing process, as applied to potential target rules in the program's functional topology. Execution of a TXL program begins by applying the rule `main` to the entire input parse tree. The main rule typically captures the highest level structure to be transformed and invokes a series of subrules (e.g. A0, B0, and C0) to perform the actual transformation. If other rules are to be applied, then these rules must explicitly invoke them, and so on, in the style of a recursive functional program.

At the bottom of Figure 2, we modify the original main rule so that the target rule B1 is directly invoked on its captured pattern tree, thereby excluding the effects of other subrules A0, B0 and C0. By doing so, we reduce the transformation to only B1 and its subrules, B2 and B3. We can implement this in TXL using a new main rule to invoke B1 alone:

```
rule main
   replace [program]
      P [program]
   by
      P [B1]
end rule
```

This approach is applicable to subrules at all levels (e.g. C2, also shown in Figure 2).

3.2 TXL Grammar Type Isolation

The basic unit of a TXL grammar is the *define* statement. Each define gives an ordered set of alternative forms (separated by the | operator) for one nonterminal type in the context-free grammar. Terminal symbols such as operators, semicolons and keywords are represented as themselves, and non-terminal types appear in square brackets `[]`. For example, the nonterminal definition:

```
define expression
      [number]
   | [expression] + [number]
   | [expression] - [number]
end define
```

specifies that an item of type *expression* is either a number, or something that is already an expression followed by a plus sign and a number, or an expression followed by a minus sign and a number.

TXL programs normally begin with a *base grammar* that forms the syntactic foundation of the transformation. The base grammar can be modified or extended by *overriding* nonterminal definitions using grammar *redefines* [4]. The effective grammar is the one formed by substituting each of the redefinitions in the order that they appear in the TXL program.

Like rules, context free grammar types in TXL are interpreted as a recursive functional program, starting with the goal nonterminal `[program]`. Our approach reduces a grammar by removing the types that functionally precede the *target definition* in the program's topology. This is realized in a simple way by overriding the grammar entry

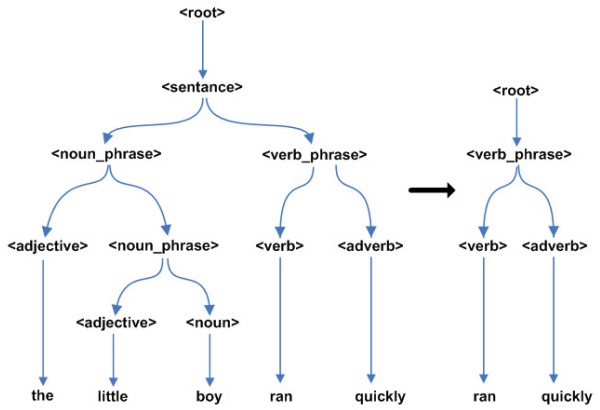

Figure 3. Grammar scope refinement to isolate parse subtrees.

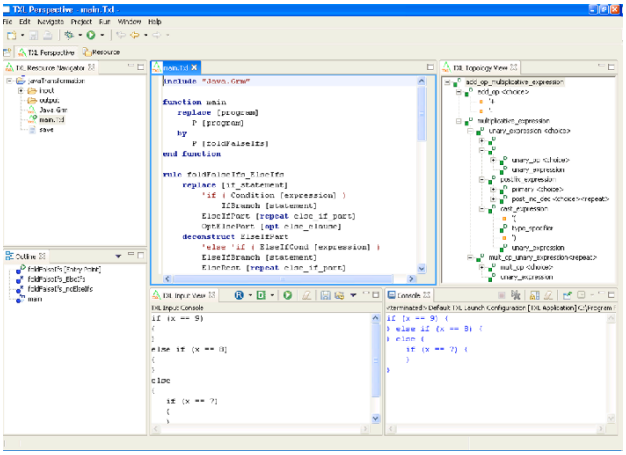

Figure 4. The TXL Perspective in Eclipse.

point, `program`, so that it directly references the nonterminal type we want to test:

```
redefine program
   [target_definition]
end redefine
```

The example in Figure 3 shows the parse tree that results when the root non-terminal is overridden by the definition of the target type `verb_phrase`.

Normally users must provide source input that meets the syntactic specifications of the entire grammar. Often this forces users to test their programs on examples that are much larger and more complex than they need to be for the purpose of testing a single rule. In conjunction with rule isolation, type isolation can greatly simplify the testing process by reducing a program's grammar to the specific pattern type of the target rule.

3.3 Non-Invasive Unit Test Automation using Eclipse

Thus far, our methodology has focused on using functional decomposition and grammar overrides to isolate individual TXL rules and types. Without an automation process, a program must be manually mutated to implement these new rule and grammar entry points. This is a cumbersome procedure, requiring developers to (i) modify the program so that it isolates the rule and/or type of interest, (ii) save the new contents in the editor, (iii) run the transformation from the command line, (iv) return to the editor to fix any compile-time errors, and finally, (v) restore the program back to its initial state, retaining any desired changes.

Even by leveraging TXL's functional semantics, this task is so time consuming that most programmers avoid doing it, even though it is the most effective way to determine program behavior [9]. Furthermore, the mutation process is invasive; users may accidentally introduce unexpected program behavior during the mutation process that is only discovered when the code is returned back to its original state.

To address these issues, we have designed TETE, a set of Eclipse plug-ins for editing, debugging and unit-testing TXL source transformations. Figure 4 shows a screenshot of the TXL Perspective provided by TETE.

Unit testing is provided by three interacting modules of TETE: a comprehensive modelling infrastructure based on the JDT framework [10] that provides a TXL-specific view of the state of the Eclipse workspace and its child resources, the Rapid Transformation Input View (RTIV), which merges rule testing, type testing, and program execution into a single, interactive interface, and the TXL Test Generator, which is responsible for automatically mutating TXL programs to isolate target rules and grammar types for testing. We discuss each of these components below. Figure 5 illustrates the automated mutation process in TETE.

TXL Model in Eclipse. The TXL Model comprises of a set of interfaces and classes that model the TXL entities associated with creating, editing, and building a TXL program. By recording model state through a central data repository, support tools are able to share a current and consistent view of the TXL program. In the context of unit-testing, the TXL Model serves to identify the file dependencies needed by the TXL Test Generator to create a temporary directory from a project's contents for non-invasive testing. It also allows the RTIV to query for the rules and types present in the TXL file being edited by the TXL Editor.

TXL RTIV. The TXL Rapid Transformation Input View (RTIV) is an interface for executing TXL programs. Whereas TETE's unit testing functionality focuses on eval-

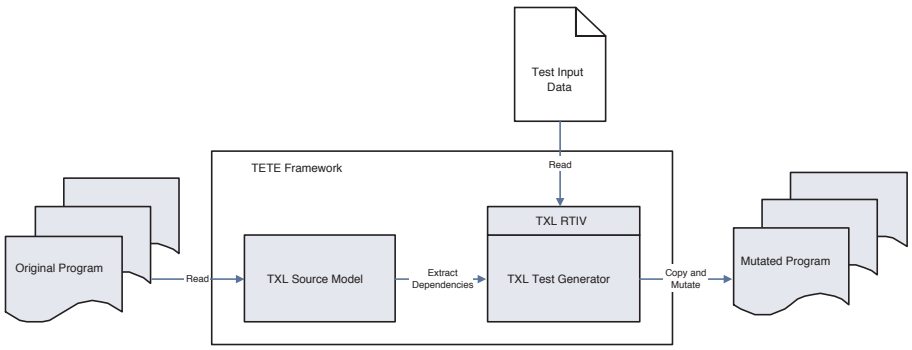

Figure 5. The TETE unit testing infrastructure is made up of the TXL Model, RTIV, and the TXL Test Generator.

uating individual program structures, the RTIV focuses on accessibility and swift interaction. It allows users to bypass the intricacies of the command line and avoid setup time to evaluate their programs. By providing a central, example-based interface for program exploration and verification, the RTIV allows users to leverage test-driven development in a piecewise, incremental, and simple manner.

The RTIV presents users with an input pane that can be used to interactively compose input for testing a program's behavior, and a toolbar that provides actions for executing and terminating a transformation, as well as for clearing, saving and loading the input. TETE's unit testing functionality is integrated into the RTIV using drop-down lists that display the rules and types available for unit testing. The lists are updated by querying the TXL Model for the source entities of the compilation unit being edited. When the user selects a new compilation unit from a different project, the lists are refreshed with the new set of rules and types.

TETE is designed to be simple and intuitive, with the aim of helping novice developers leverage the pedagogic benefits of unit testing without being distracted by an overly complex interface. In particular, the RTIV provides a lightweight style of interaction with a minimal number of controls. Figure 6 shows the step-by-step process used to compose and execute a TXL unit test in the RTIV.

To run a TXL experiment, a user selects a target rule or target grammar type from the respective RTIV drop-down menu, types or loads the test data into the RTIV input view, and executes the transformation by pressing the RTIV's launch button. The composed input is saved to a temporary file, and directed to the TXL interpreter as the transformation input file. The contents of the rule and type drop-down lists are dependent on the active TXL compilation unit that is displayed in the editor. When the user executes the transformation, the output and error streams are directed to the standard Eclipse Debug Console for display.

TXL Test Generator. The TXL Test Generator provides the program launching infrastructure responsible for automating program mutations in a non-invasive manner. Implementing the mutations necessary for unit testing types and rules directly in the Eclipse workspace would cause several unwanted side-effects. Firstly, it would be error-prone and confusing for programmers, because the testing version of the source code no longer reflects the semantics of the original program. TXL compilation units and their working copies should only reflect those changes that are committed by the user. Secondly, it would force unnecessary updates to the TXL Model, when program modifications are only necessary for the duration of the test execution. For these reasons, TETE exposes unit testing mutations to the TXL processor only, leaving the workspace artifacts themselves unmodified.

To implement this separation, a "shadow" copy of the test program's project in Eclipse is created in a temporary directory on the local file system. The program's main file, containing the original main rule, is copied to this directory and mutated by the test generator to reflect the modified transformation entry point. By working on this test copy we ensure that unit testing mutations are non-invasive.

The relative source location and range of the code in the generated test copy must remain consistent with the corresponding code in the original version. Modifying the source offset and range of the code may cause incorrect behavior during error reporting, because syntactic errors reported by TETE will refer to locations in the generated source code instead of the actual locations reflected in the TETE workspace. By keeping the original source contents of the workspace file in an equivalent contiguous space in the test file, we ensure that source locations in the original files are preserved even after modifications are made to the corresponding shadow copies, and thus error reporting remains consistent between the test and workspace versions.

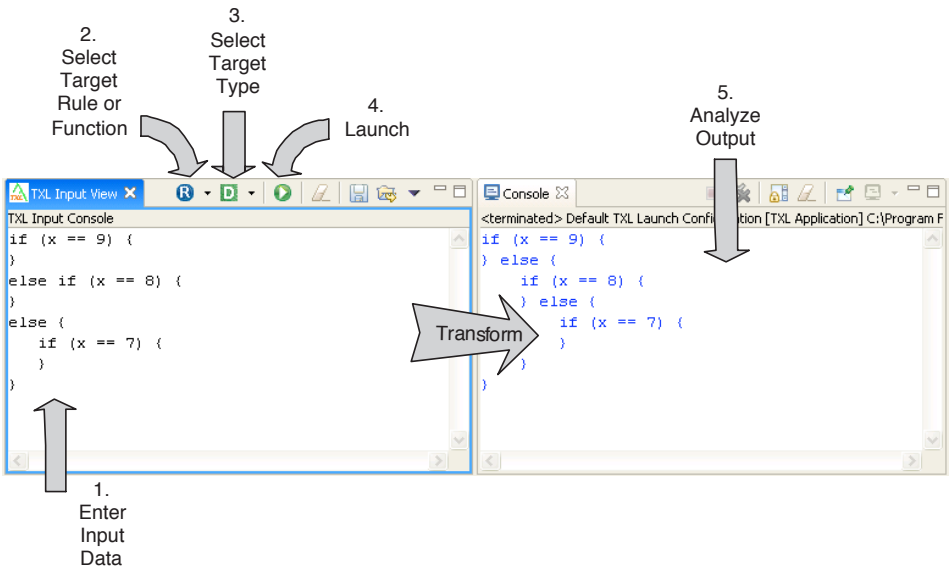

Figure 6. RTIV view for program unit test composition and launching.

4 An Example

The remainder of the paper demonstrates the use of TETE's automated unit testing interface using the example program shown in Figure 7. The goal of this program is to fold Java if-statements with the constant *false* as the branching condition. In this example, TETE's unit testing infrastructure is used to test and debug the program's rules and application strategy.

In C++ and other object-oriented languages, test cases are represented by examples of method invocations on target class members. In TXL, test cases are represented by examples of input source from the problem domain targeting specific subtransformations and grammatical types.

Figure 8 shows two example test cases that should be handled by our program, as well as the expected transformed results for each. In the first case, the target input contains a false if-statement followed by a series of else-ifs (Figure 8(a)). The program must handle this case by folding out the if-branch and propagating the first else-if branch upwards (Figure 8(b)). In the second test case, if the false if-statement is followed immediately by an else branch (Figure 8(c)), then the entire statement group is replaced by the body of the else-branch (Figure 8(d)). To keep the demonstration simple we'll only consider these two. In practice, users would develop many other test cases to convince themselves of the transformation's coverage.

The approach uses two primary TXL subrules, `foldFalseIf_ElseIfs` and `foldFalseIf_noElseIfs`, which search for and optimize false if-statements with and without else-if branches respectively. These embedded subrules are invoked by the rule `foldFalseIfs`, which encodes an application strategy that applies `foldFalseIf_noElseIfs` first and `foldFalseIf_ElseIfs` second. `foldFalseIfs` is invoked on the entire program scope captured by the main rule.

As a first attempt to evaluate the correctness of our transformation, we enter the input data shown in Figure 8(a) into the RTIV's input pane and launch our test, using the default main function as the entry point. The RTIV returns with the transformed output:

```
class C
{
  public void foo(int x) {
      x = 0;
  }
}
```

which is incorrect compared to our expected output in Figure 8(b).

So far, the evaluation process has mirrored that of most other transformation systems. We have executed our program using a black box test in which composite subtransformations are opaquely merged into the final result. In the absence of unit testing, we must continually apply this coarse-grained execution approach so that the entire transformation is repeatedly modified and re-executed until it returns with the expected output.

Given the erroneous output, we are unsure of whether the error is caused by a faulty pattern-replacement in one

```
include "Java.Grm"

function main
  replace [program]
    P [program]
  by
    P [foldFalseIfs]
end function

rule foldFalseIfs
  replace [statement]
    S [statement]
  construct NewS [statement]
    S [foldFalseIfs_noElseIfs]
      [foldFalseIfs_ElseIfs]
  deconstruct not NewS
    S
  by
    NewS
end rule

rule foldFalseIfs_noElseIfs
  replace [statement]
    'if (false)
       IfBranch [statement]
    OptElsifs [repeat else_if]
    'else
       ElseBranch [statement]
  by
    ElseBranch
end rule

rule foldFalseIfs_ElseIfs
  replace [statement]
    'if (false)
       IfBranch [statement]
    'else 'if (ElseIfCond [expr])
       ElseIfStatement [statement]
     ElseRest [repeat else_if_part]
    OptElsePart [opt else_clause]
  by
    'if ( ElseIfCond )
       ElseIfStatement
    ElseRest
    OptElsePart
end rule
```

Figure 7. A TXL program to fold if-statements containing the literal 'false' in the condition.

```
class C
{
  public void foo(int x) {
    if (false){
       x = 10;
    }
    else if (x == 0) {
       x = 1;
    }
    else {
       x = 0;
    }
  }
}
```

(a) Input with Else-ifs

```
class C
{
  public void foo(int x) {
    if (x == 0) {
       x = 1;
    }
    else {
       x = 0;
    }
  }
}
```

(b) Folded output from (a)

```
class C
{
  public void foo(int x) {
    if (false){
       x = 10;
    }
    else {
       x = 0;
    }
  }
}
```

(c) Input with no Else-ifs

```
class C
{
  public void foo(int x) {
     x = 0;
  }
}
```

(d) Folded output from (c)

Figure 8. Two example test cases for false if-statement folding.

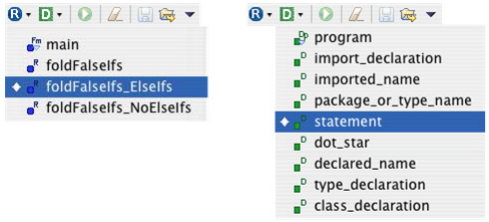

Figure 9. RTIV rule and type selection lists for unit testing.

```
rule foldFalseIfs
  replace [statement]
    S [statement]
  construct NewS [statement]
    S [foldFalseIfs_ElseIfs]
      [foldFalseIfs_noElseIfs]
  deconstruct not NewS
    S
  by
    NewS
end rule
```

Figure 10. Application order modification made to original program.

of the subrules, or whether the program's rule application strategy is incorrect. Since our test input contains a leading if-statement with an else-if branch, we decide to directly test the rule that is responsible for this type of pattern, `foldFalseIfs_ElseIfs`.

One of the complicating factors of our current execution setup is that the test input listed in Figure 8(a) is for the entire program scope of our main function, instead of for the rule we want to test, `foldFalseIfs_ElseIfs`. We can simplify our test to directly reflect the pattern scope of the target rule, which is of type `statement`. To do this we must ask TETE to isolate the target type `statement` and the target rule `foldFalseIfs_ElseIfs`. We select these constructs in the RTIV rule and type drop-down lists, as shown in Figure 9.

Now we can enter a test case consisting of just the if statement into the RTIV's input pane:

```
if (false) {
   x = 10;
}
else if (x == 0) {
   x = 1;
}
else {
    x = 0;
}
```

and run the `foldFalseIfs_ElseIfs` rule alone on it.

When we do so, TETE returns with the correct, transformed output listed in Figure 8(b). By testing the subrule `foldFalseIfs_ElseIfs` directly, we have verified its correct operation, effectively eliminating the possibility of an erroneous pattern-replacement in this rule. The next step in our evaluation process is to test the rule that invokes the one we have just tested, `foldFalseIfs`.

This rule implements an application strategy in which all if-statements without else-if branches (those handled by `foldFalseIfs_noElseIfs`) are transformed before those that do (handled by `foldFalseIfs_ElseIfs`).

As a next try, we conduct a new rule test, this time by selecting `foldFalseIfs` from the RTIV's rule drop-down list. Again, we launch the test, but this time we receive the original erroneous result, demonstrating that the problem originates in `foldFalseIfs`.

The cause of this error can be explained in terms of the rewrite behavior of the two subrules called by `foldFalseIfs`: both `foldFalseIfs_noElseIfs` and `foldFalseIfs_ElseIfs` accept the same pattern scope. Whereas `foldFalseIfs_ElseIfs` simply propagates the highest else-if statement into the if-branch position, `foldFalseIfs_noElseIfs` replaces its entire scope by the contents of the else-block if it matches a false if-statement.

By calling `foldFalseIfs_noElseIfs` before `foldFalseIfs_ElseIfs`, every else-if statement that follows a false if-branch is removed by the time `foldFalseIfs_ElseIfs` is invoked. While further unit testing will eventually reveal that the pattern of `foldFalseIfs_noElseIfs` is overly aggressive, a more immediate solution to this error is to simply rearrange the application order in `foldFalseIfs`, so that `foldFalseIfs_ElseIfs` is invoked first (Figure 10). Thus, all the else-if statement manipulation is completed on `foldFalseIfs`'s replacement tree before it is passed to `foldFalseIfs_noElseIfs`.

After making this change, we save the contents of the TXL Editor and re-execute the unit test for `foldFalseIfs`, yielding the expected result:

```
class C
{
  public void foo(int x) {
    if (x == 0) {
       x = 1;
    }
    else {
       x = 0;
    }
  }
}
```

We can continue testing with a variety of different inputs (including the test case in Figure 8(c)) to uncover and debug other problems using unit testing in similar fashion, until we are confident that we don't need any further changes.

TETE provides many other facilities to assist in testing and debugging TXL programs, including saving and loading unit test cases, rule and grammar navigation panels, configurations for larger test inputs, and so on. While we have concentrated on TXL in this work, we believe that similar methods can easily be used with other source transformation languages and tools such as those below.

5 Related Work

While TXL has its own way of expressing source transformations, many other tools and languages share similarities. Capturing instances of structured syntax, or *patterns*, and replacing them with alternate structures, or *replacements*, is a fundamental concept in transformation-capable tools. These tools typically differ in their methods of rewrite and grammar specification and interpretation.

The ASF+SDF Meta-Environment [12, 13] is a generic development framework that supports automatic construction of interactive tools for programs and specifications written in a formal language. ASF+SDF is a modular, first-order formalism for the integrated specification of syntax and semantics. SDF provides the syntax definition component and ASF uses re-write rules to describe transformations. ASF+SDF supports a fixed number of non-programmable tree-traversal strategies (i.e. top-down, bottom-up) to visit the nodes of a parse tree in a predictable order.

Stratego/XT [14, 15, 16] is a hybrid language and toolset framework that supports the specification of program transformation systems based on the paradigm of rewriting strategies, which augment rewrite rules with generic strategies for their application. Stratego/XT provides a strategy unit-testing tool called SUnit that allows for the specification of tests to apply a strategy to a specific term and compare the result to the expected output. This idea is similar to rule testing in TETE, except that program traversal in TXL is an inherent part of functional decomposition. Stratego/XT also provides a command-line tool for testing SDF grammar definitions, similar to TETE's grammar type testing. Tests written with SUnit and ParseUnit must be specified in separate test suites from the command line, whereas TETE merges rule, strategy and grammar testing together in a single, comprehensive interactive interface.

The DMS Software Reengineering Toolkit (SRT) [1, 2] is a commercial program analysis and transformation system based on user-configurable, generalized compiler technology. It includes facilities for defining language syntax and deriving context-free parsers and pretty-printers for languages such as C, C++, COBOL, and Java. SRT is capable of defining multiple, arbitrary specification and implementation languages (or *domains*), as well as applying transformations to source code written in any combination of the defined domains.

Extensible Stylesheet Language Transformations (XSLT)[3, 17] is the W3C standard for the source transformation of XML documents. While XSLT is not a general-purpose transformation technology, it shares several properties with current source transformation languages such as Stratego/XT, ASF+SDF, and TXL, such as its pure-functional, user-programmable paradigm, and its use of pattern match and replacement pairs applied in a term-by-term rewriting style.

6 Future Work

The TETE framework provides a proof of concept for introducing isolated rule and grammar execution to the TXL source transformation paradigm. By leveraging TXLs functional semantics in combination with the source infrastructure of Eclipse, transformations can be non-invasively unit tested using a rule-by-rule, type-by-type approach.

Thus far, TETE is still in the proof-of-concept stage and many issues need to be faced as it comes into practice, particularly if it is to be used for experienced as well as novice users of TXL. TETE has a number of limitations that need to be addressed, for example:

At present only unparameterized rules are supported by TETEs unit testing infrastructure. TXL subrules may optionally be parameterized by arguments captured in patterns of the invoking rule. To support unit testing of parameterized rules, TETEs source model must be extended to identify and connect rule parameters with their associated rules.

A production unit testing facility should automate the loading, running and verification of suites of tests rather than just one at a time. Presently, each unit test in TETE must be loaded and run separately. We would like to extend TETEs unit testing interface so that users can configure and run multiple tests in unison, and save test suites to support regression testing.

TETE does not presently support the TXL stepwise debugger (TXLDB). If it is to be used in debugging complex rulesets, provision for stepwise execution and back-tracing of rule applications is required.

Like all new interfaces, TETE requires feedback from practical use to identify unexpected use cases and tune the interface and work flow to match the way TXL programmers actually develop and test transformations. While TETE is already being used as a convenient TXL editor, it is yet to be used for its original intent: helping novices explore and test TXL programs as an aid to understanding. Our plan for this year is to introduce it to undergraduate and graduate classes using source transformation.

7 Conclusion

Unit testing has many benefits that have been well documented in the procedural and object-oriented programming paradigms. Automated unit testing for source transformation is a much more recent concept. We have identified a list of five paradigm-specific motivations for introducing unit testing to source transformation programming.

By leveraging TXL's functional decomposition semantics in combination with Eclipse's scalable language-modelling framework, TETE automates rule and grammar unit testing and allows users to more easily develop, debug, and analyze their source transformations in a piecewise manner.

References

[1] L. Aversano, M. D. Penta, and I. D. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proc. SCAM 2002*, pages 83–92, 2002.

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

[3] J. Clark. XSL Transformations (XSLT) version 1.0. (http://www.w3.org/tr/xslt), 1999.

[4] J. R. Cordy. TXL - A Language for Programming Language Tools and Applications. *Proc. LDTA 2004*, pages 1–27, April 2004.

[5] J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language v10.4*. School of Computing, Queen's University, January 2005.

[6] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source Transformation in Software Engineering using the TXL Transformation System. *J. Information and Software Technology*, 44(13):827–837, 2002.

[7] S. H. Edwards. Rethinking Computer Science Education from a Test-First Perspective. In *Companion to OOPSLA '03*, pages 148–155. ACM Press, 2003.

[8] S. H. Edwards. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proc. SIGCSE '04*, pages 26–30. ACM Press, 2004.

[9] C. Reis. A Pedagogic Programming Environment for Java that Scales to Production Programming. Master's thesis, Rice University, April 2003.

[10] J. C. team. Java development tooling core. Technical report, JDT Core team, May 2005. http://www.eclipse.org/jdt/.

[11] O. Technologies. Eclipse Platform Technical Overview. Technical report, Object Technology International, Inc., Feb. 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[12] M. G. J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual*. Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, January 2005.

[13] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[14] E. Visser. The Stratego Reference Manual 0.5 edition (http://www.stratego-language.org/doc/reference/html/index.html).

[15] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proc. RTA '01*, pages 357–362, London, UK, 2001. Springer-Verlag.

[16] 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.

[17] P. Wadler. A Formal Semantics of Patterns in XSLT and XPath. *Markup Lang.*, 2(2):183–202, 2000.

[18] H. Younessi, P. Zeephongsekul, and W. Bodhisuwan. A general model of unit testing efficacy. *Software Quality Control*, 10(1):69–92, 2002.

[19] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.