SANDIA REPORT

SAND2004-0868
Unlimited Release
Printed March 2004

Dependable Software through Higher-order Strategic Programming

Victor Winter
Sandia Contract No. 5137
University of Nebraska at Omaha

Steve Roach
University of Texas at El Paso

Fares Fraij
University of Texas at El Paso

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-0868
Unlimited Release
Printed March 2004

Dependable Software through Higher-order Strategic Programming

Victor Winter
Sandia Contract No. 5137
University of Nebraska at Omaha
Department of Computer Science
vwinter@mail.unomaha.edu

Steve Roach
University of Texas at El Paso
Department of Computer Science
sroach@cs.utep.edu

Fares Fraij
University of Texas at El Paso
Department of Computer Science
fzfraij@utep.edu

Abstract

Program transformation is a restricted form of software construction that can be amenable to formal verification. When successful, the nature of the evidence provided by such a verification is considered strong and can constitute a major component of an argument that a high-consequence or safety-critical system meets its dependability requirements.

This article explores the application of novel higher-order strategic programming techniques to the development of a portion of a class loader for a restricted implementation of the Java Virtual Machine (JVM). The implementation is called the SSP and is intended for use in high-consequence safety-critical embedded systems. Verification of the strategic program using ACL2 is also discussed.

Contents

1 Introduction **7**
1.1 Impact of Transformation on Dependable Software Construction . . . 7
1.2 Contribution . . . 7

2 Background **8**
2.1 Equational Reasoning: The Foundation of Transformation . . . 8
2.2 Strategic Programming . . . 9
2.3 The Distributed Data Problem . . . 10

3 An Overview of TL **10**
3.1 Terms and Trees . . . 11
3.2 Match Equations and Match Expressions . . . 11
3.3 Rewriting in TL . . . 12
3.4 The Basics of TL . . . 13
3.5 Higher-Order Strategies in TL . . . 14
3.6 TL's *transient* Combinator . . . 16

4 A Class Loader for Java **17**
4.1 Method Table Requirements for the SSP . . . 17
4.2 An Example of Method Table Construction . . . 18
4.3 Constructing SSP Method Tables in TL . . . 20

5 Assurance **23**

6 Related Work **25**

7 Conclusion **26**

List of Figures

1 A Simple Equational Theory . . . 8
2 A source-to-source optimization . . . 8
3 Grammar fragment for a simplified Java class file . . . 11
4 The parse trees corresponding to *classfile*[[$\{class_1\ super_1\ methods_1\}$]] and *classfile*[[$\{id_1\ id_2\ methods_1\}$]] . . . 12
5 The semantics of sigma distribution . . . 12
6 The primitive constructs, combinators and constants of TL . . . 14
7 Some basic first-order traversals of TL . . . 14
8 Diagram of a *tdl-broadcast* traversal from the perspective of strategy application . . . 15
9 Diagram of a *tdl* traversal from the perspective of strategy application . . . 15
10 A small abstract grammar . . . 16

11 A small grammar 17
12 A left-biased composition of three transient strategies 17
13 The structure of the term *item_list*[[*b b b*]] 18
14 A simple application 19
15 Abstract method table entries for the classes A, B, C, and D 19
16 A simplified grammar for class files 20
17 Strategies for creating a class hierarchy 22
18 The sequential composition of transient strategies 22
19 The strategy remaining after inserting cf_A into the children list of *obj*. 22
20 The strategy remaining after inserting cf_B and cf_C into the children list of cf_A. 22
21 Strategies for inserting methods into method tables 22

Nomenclature

ACL2	A Computational Logic in Applicative Common Lisp
BNF	Backus-Naur Form
HATS	High Assurance Transformation System
JVM	Java Virtual Machine
ROM	Read Only Memory
SSP	Sandia Secure Processor
TL	Transformation Langauge – a higher-order strategic programming language

1 Introduction

The manipulation of software artifacts such as specifications and program source code through rewriting is an active area of research [3][5][7][13][14][23][27][24][30]. Driving this research is the idea that the repeated application of a set of simple rewrite rules can effect a major change in a software artifact. Within the scope of this article we will use the term *program transformation* (or *transformation*) in a general sense to refer to software manipulation processes that are restricted to the fully automatic application of rewrite rules. We will also predominantly refer to the objects that are the subject of transformation as *terms* rather than specifications, programs, code fragments, or other artifacts.

1.1 Impact of Transformation on Dependable Software Construction

Theoretically speaking, the use of transformation for the maipulation of software artifacts can span the entire software life cycle from the derivation of implementations from formal specifications to software maintenance and reverse engineering. From a practical perspective, the automatic application of rewrite rules to alter programs provides the foundation for scaling transformation-based software development methods to large systems. From the perspective of dependability, the explicit nature of transformation exposes the software development process to various forms of analysis that would otherwise not be possible. In contrast to the "programmer at the terminal" software development paradigm where the thinking process of the programmer is not explicit, the development of software through transformation is a repeatable process in the sense that a third party can replay the transformation sequence used to develop the software. The notion that the rewrite rules used within a transformation should be "simple" also provides hope that formal verification efforts could succeed in proving that the rules that are applied during the course of a transformation preserve correctness. When successful, the nature of the evidence provided by such a formal verification is considered strong and can constitute a major component of an argument that a high-consequence or safety-critical system meets its dependability requirements.

1.2 Contribution

This paper explores the novel use of higher-order strategic programming techniques available in the strategic programming language TL [29] to the development of a portion of a class loader for a hardware implementation of the Java Virtual Machine(JVM). The implementation under consideration is the Sandia Secure Processor(SSP) developed at Sandia National Laboratories for use in high-consequence safety-critical systems. In this article, we will take an in-depth look at how TL's novel higher-order strategies, traversals, and *transient* combinator can be used to construct the method tables required by the SSP. We also provide a sketch of how such strategies might be verified using ACL2. We believe that the abstractions provided by TL's higher-order strategies positively impact the ability of an automated verification system such as ACL2 to successfully prove the correctness of a variety of strategic programs, thereby contributing to the development of dependable software systems.

The remainder of the paper is as follows: Section 2 provides background on program transformation. Section 3 gives an overview of the higher-order strategic programming language TL. Section 4 briefly describes the SSP and gives a detailed discussion of how method tables for the SSP can be constructed

in TL. Section **??** briefly describes a transformation system we are developing that supports a restricted dialect of TL. All problems mentioned and presented in this article have been implemented in TL. Section 5 describes our research in verification. Section 6 describes related work in the areas of strategic programming and transformation verification, and Section 7 concludes.

2 Background

This section provides the background on program transformation. We briefly discuss why transformation is considered promising in the context of dependable software development as well as some fundamental problems one faces when using this paradigm.

2.1 Equational Reasoning: The Foundation of Transformation

Equational reasoning lies at the heart of transformation-based software development. An *equational theory* is a set of equations that contain the knowledge necessary to realize a desired transformational objective. The knowledge embodied in an equational theory can capture a wide variety of properties about software artifacts and application domains including (1) relationships between domain-specific data types and implementation-specific data types, (2) general relationships between specification-level and implementation-level constructs, and (3) source-to-source equivalences such as optimizations.

Figure 1 is an example of a simple equational theory consisting of equalities derived from the formal semantics of a typical imperative programming language. The equalities of this equational theory can be used to perform the *transformation* (denoted by the symbol $\Rightarrow$) shown in Figure 2.

while (false) do block	=	skip
skip ; statement	=	statement
id < id	=	false

Figure 1: A Simple Equational Theory

Original Code Fragment		Transformed Code Fragment
...		...
s_1;		s_1;
while $(x < x)$ do { ... };	$\Rightarrow$	
s_2;		s_2;
...		...

Figure 2: A source-to-source optimization

When using equational theories as the basis of transformation, a problem that must be solved is that of controlling the application of equations in order to realize a given objective. We refer to this problem as the *control problem*. Ideally, one would like the control problem to be solved automatically (e.g., by the searching capabilities of the computer). In such an approach, the computer would continue to apply equational reasoning steps until the given transformational objective has been reached. Unfortunately,

such an approach is in general undecidable. Rewriting systems address this problem by providing restrictions on the application of equational reasoning steps that, under the right conditions, result in a set of rules for which the control problem is decidable [1]. The basic idea behind rewriting is to orient equalities so that (1) equational reasoning steps (i.e., the substitution of equals for equals) can only be applied in one direction and (2) the resulting rule set is *confluent*, that is, the order of application of the equations does not affect the final result. This restriction on the direction of equational reasoning is expressed by replacing the symmetric equality relation $=$ with the anti-symmetric rewrite relation $\rightarrow$. For example, the equation $s = t$ states that the terms s and t may be freely substituted in place of one another. In contrast, the rewrite rule $s \rightarrow t$ states that s may be replaced by t, but it does not specify that t may be replaced by s.

In a rewriting framework, the difficulties associated with the control problem are shifted to the problem of properly orienting equalities within an equational theory and producing rule sets that are confluent. Another desirable property of a rule set is that it is *terminating*, meaning that the rewriting process is guaranteed to stop. If a rule set can be shown to be both confluent and terminating, then the exhaustive application of rules in any order will always yield the same result, also known as a *normal form*. Under these conditions, the solution to the control problem becomes trivial.

2.2 Strategic Programming

Unfortunately, the manipulation of software artifacts generally gives rise to rule sets that are not and cannot be made terminating or confluent. One approach for dealing with such rule sets is to introduce additional function symbols into the term language for the explicit purpose of controlling the application of rewrite rules. This approach yields sets of rewrite rules that are not very reusable and is met with considerable resistance when dealing with the scale and complexity found in real-world languages [5]. In an alternate approach called strategic programming [14], the control problem is solved by making explicit a number of *combinators* capable of specifying both which rewrite rules should be applied to a given term as well as to which sub-terms a rewrite rule should be applied. These combinators can be used together with rewrite rules to form expressions called *strategies*. In this context, rewrite rules themselves are also referred to as strategies.

The most common combinators controlling which rules should be applied to a given term are those that enable the sequential or conditional composition of rewrite rules to be expressed. Let $(s)t$ be the term resulting from the application of strategy s to term t. A semi-colon is used to denote the sequential composition combinator, and the symbol $<+$ is used to denote the *left-biased choice* combinator. For example, the strategic expression $s_1; s_2$ denotes the sequential composition of the strategy s_1 and s_2. Its evaluation will first apply s_1 to t followed by the application of s_2. Similarly, the strategy $s_1 <+ s_2$ denotes the left-biased composition of s_1 and s_2. The expression $(s_1 <+ s_2)t$ will apply s_1 to t, and if this application is successful the resulting term will be returned, otherwise s_2 will be applied to t and its result returned.

Traversals are strategies that specify where rules should be applied within a term structure. Specifically, traversals define which sub-terms in a term should be visited as well as the order in which sub-terms are visited. A *generic traversal* is a general purpose traversal that can be applied to an arbitrary term structure. Though the names may vary between strategic systems, classic examples of generic traversals include TDL, which traverses a term structure in a top-down left-to-right fashion (i.e., outside-in),

and BUL, which traverses a structure in a bottom-up left-to-right fashion (i.e., inside-out). Generic traversals such as TDL and BUL are parameterized on other strategies. For example, if s denotes a strategy (e.g., a rewrite rule) developed for a particular problem domain, then the expression $TDL(s)t$ will traverse t in a top-down left-to-right fashion and attempt to apply s to every term encountered.

2.3 The Distributed Data Problem

Rewrite rules and strategies are highly effective at manipulating a software artifact when the information necessary for the manipulation can be captured via standard matching or unification. Unfortunately, transformational objectives often require the manipulation of non-local information. For example, the type of a declared variable might need to be distributed to all the places where that variable occurs within a program. Similarly, in order to rename a variable, a unique *id* is generated and this *id* must then replace all occurrences of the original variable in the program. In Winter [29], this kind of non-localized data exchange between terms has been characterized and is referred to as the *distributed data problem.*

Most strategic systems in use today are first-order: rewrite rules may only be applied to terms, and the successful application of a rewrite rule to a term yields another term. In these systems the application of a rewrite rule to a term may not yield another rewrite rule. In a first-order setting, a combination of data accumulation and parameter passing is the standard approach taken to solve the distributed data problem. This typically involves the creation of auxiliary structures such as lists to store data that is to be distributed as well as accompanying lookup functions to extract data from such lists. Strategy parameterization is used as the mechanism by which these lists are distributed to the appropriate parts of a term.

In this paper we describe a novel solution to the distributed data problem that is based on higher-order strategies. The idea is to accumulate data in the form of a strategy rather than an auxiliary structure such as a list. This conceptual shift is motivated by the observation that strategy application is a primitive operation in a strategic framework, as are the combinators used to construct such strategies. The use of lists, lookups,and parameterization can be subsumed by dynamic strategy creation and application. As a result, such higher-order strategies provide an elegant technique for solving instances of the distributed data problem.

We have developed and implemented a higher-order strategic language called TL [29], which we are using to explore the implications of higher-order strategies on software development. We have also developed an IDE for strategic programming called HATS that is based on TL. HATS [10] is platform independent and freely available. All examples presented and mentioned in this paper have been implemented in HATS.

3 An Overview of TL

TL is an *identity-based* higher-order strategic language for rewriting *parse trees.* When a rule or strategy fails to apply to a term, the term is returned unchanged. In contrast, most strategic systems are *failure-based* because they will return a special value *fail* when a strategy fails to apply to a term. In TL, a domain (i.e., a term language) is defined using an Extended-BNF notation. Terms, or *parse trees*, are

classfile	::=	"{" class super methods "}"
class	::=	id
super	::=	id
id	::=	identifier

Figure 3: Grammar fragment for a simplified Java class file

described using a special abbreviated notation and are called *parse expressions*. This section gives an overview of TL.

3.1 Terms and Trees

While most strategic languages define term structures using abstract syntax, we use parsing technology to define term structures. Thus, when we say "structure" or "term" what we really mean is a parse tree. For a given grammar, we will write $B[[\alpha']]$ to denote a parse tree corresponding to the derivation $B \overset{+}{\Rightarrow} \alpha$ where any nonterminals occurring in α have been subscripted yielding α'. In general, we will use the term *parse expression* to refer to expressions of the form $B[[\alpha']]$.

When viewed from the perspective of parse trees the derivation $B \overset{+}{\Rightarrow} \alpha$ denotes a tree whose root is B and whose leaves are α. Nonterminals when they occur in leaf positions (i.e., in α) are subscripted so they may be distinguished from one another. A subscripted nonterminal is called a *schema variable* or simply a *variable*. Parse expressions containing no schema variables are referred to as being *ground*.

Consider the BNF grammar fragment shown in Figure 3 defining the structure of a simplified Java class file consisting of a class name, a super class name, and a methods section. A term describing such a class structure could be written as follows:

$$\mathit{classfile}[[\ \{\mathit{class}_1\ \mathit{super}_1\ \mathit{methods}_1\}\ ]]$$

Note that this term structure has a derivation length of one, since the nonterminal *classfile* directly derives $\{\mathit{class\ super\ methods}\}$. We would like to point out that terms may also correspond to derivation sequences whose length are greater than one. For example, we could also write:

$$\mathit{classfile}[[\ \{\mathit{id}_1\ \mathit{id}_2\ \mathit{methods}_1\}\ ]]$$

The parse trees for both of the terms discussed are shown in Figure 4.

3.2 Match Equations and Match Expressions

Matching is a fundamental operation in our framework. We will use the symbol $\ll$ adapted from the ρ-calculus [7] to denote first-order matching modulo an empty equational theory. Let t_2 denote a ground parse expression, and let t_1 denote a parse expression that may contain one or more schema variables. The equation $t_1 \ll t_2$ is a match equation. (Equivalently we may also write $t_2 \gg t_1$.) A match equation is a boolean valued operation that produces a substitution σ as a by-product. A substitution σ binding schema variables to ground parse expressions is a solution to $t_1 \ll t_2$ if $\sigma(t_1) = t_2$ with $=$ denoting

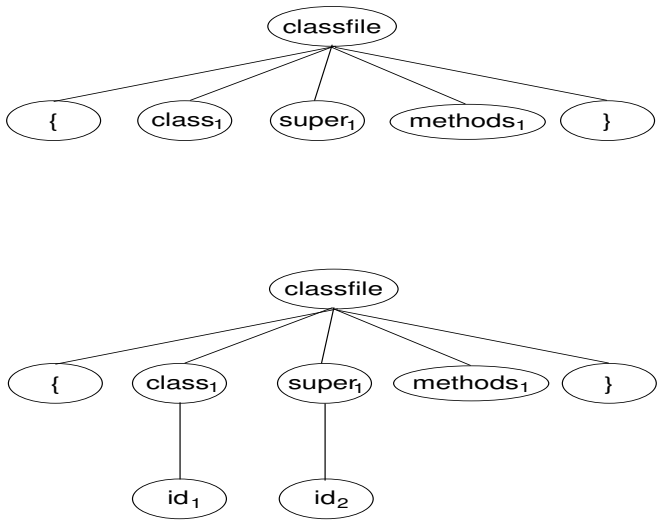

Figure 4: The parse trees corresponding to *classfile*[[$\{class_1\ super_1\ methods_1\}$]] and *classfile*[[$\{id_1\ id_2\ methods_1\}$]]

$\sigma(e_1 \wedge e_2)$	$\stackrel{def}{=}$	$\sigma(e_1) \wedge \sigma(e_2)$
$\sigma(e_1 \vee e_2)$	$\stackrel{def}{=}$	$\sigma(e_1) \vee \sigma(e_2)$
$\sigma(\neg e_1)$	$\stackrel{def}{=}$	$\neg(\sigma(e_1))$
$\sigma(t_1 \ll t_2)$	$\stackrel{def}{=}$	$\sigma(t_1) = t_2$
$\sigma(t_1 \gg t_2)$	$\stackrel{def}{=}$	$t_1 = \sigma(t_2)$

Figure 5: The semantics of sigma distribution

a boolean valued test for syntactic equality. When a match equation has a solution, the value of the match equation is the boolean value *true*.

A *match expression* is a boolean expression involving one or more match equations and forms the conditional portion of a rewrite rule (see Section 3.3). Match expressions may be constructed using the standard boolean operators: $\wedge, \vee, \neg$. A substitution σ is a solution to a match expression m iff $\sigma(m)$ evaluates to true using the standard semantics for boolean operators in conjunction with the semantics defined in Figure 5.

3.3 Rewriting in TL

A basic first-order rewrite rule has the form:

$$r : lhs \rightarrow rhs \textbf{ if } condition$$

where lhs and rhs denote terms as defined in the previous section, $lhs \rightarrow rhs$ denotes a body, r denotes an optional rule label, and *condition* denotes an optional match expression that a rule must satisfy

before it can be successfully applied.

The *application* of a rule to a term has the same syntax as the application of a function to an argument. For example, the expression $r(t)$ denotes the application of the strategy r to the term t. We will only consider the application of rules to ground terms (i.e., terms exclusively having terminal symbols in leaf positions). Given this restriction, the application of the rule r to the term t can be accomplished by discovering a substitution σ such that $lhs \ll t$ is true while simultaneously satisfying the *condition* associated with the r. If this can be accomplished, then the rule application is said to *succeed* and the term $\sigma(rhs)$ replaces t. On the other hand, if lhs cannot be matched with t or if the condition associated with the rule cannot be satisfied, then the rule application is said to *fail*. In TL, when a rule application $r(t)$ fails, the term t is returned unchanged.

When expressing strategy application, we adopt a curried notation in the functional style of ML where strategy application is left-associative and parenthesis can be used to override precedence or may be optionally included to enhance readability. For example, $r\ t$ denotes the application of r to t and has the same meaning as $r(t)$.

The body of a basic second-order rewrite rule has the form:

$$lhs_2 \rightarrow lhs_1 \rightarrow rhs_1$$

where the $\rightarrow$ symbol is right-associative. The application of the second order strategy $lhs_2 \rightarrow lhs_1 \rightarrow rhs_1$ to a term t will yield a first-order strategy of the form $\sigma(lhs_1) \rightarrow \sigma(rhs_1)$ where σ is a substitution that matches lhs_2 and t. On the other hand, if the application fails, then the strategy *skip* is returned as the result of the application. The *skip* strategy never succeeds. Its use is described in Section 3.6. The strategy *skip* has the following properties:

$skip; s$	$\equiv$	s
$s; skip$	$\equiv$	s
$skip <\!\!+\ s$	$\equiv$	s
$s <\!\!+\ skip$	$\equiv$	s
$skip +\!\!>\ s$	$\equiv$	s
$s +\!\!>\ skip$	$\equiv$	s

3.4 The Basics of TL

Figure 6 lists most of the primitives of TL. Figure 7 gives a few of the first-order generic traversals that are commonly used when constructing TL strategies.

The traversal *TDL_BR* is a *broadcasting* traversal. Broadcasting traversals are unique to TL and enable transient strategies to be controlled in interesting ways (see Section 3.6). The evaluation of the strategic expression $TDL_BR(s)t$ will first apply the strategy s to the term t. In the most general case, the result of such an application will alter both s as well as t (see Section 3.6). Let s' denote the strategy resulting from the application of s to t. Since *TDL_BR* is a broadcasting traversal, a distinct copy of s' will be applied to each of the sub-terms of t. Figures 8 and 9 respectively show the behavior of a *tdl_broadcast* and *tdl* traversal.

skip	A strategy constant that never applies.
$lhs \rightarrow rhs$ if *condition*	A conditional first-order strategy.
$lhs \rightarrow s^n$ if *condition*	A conditional strategy of order $n+1$.
$s_1^n; s_2^n$	Sequential composition: first apply s_1 and then apply s_2.
$s_1^n <+ s_2^n$	Left-biased choice: first try to apply s_1 and if that fails then try to apply s_2.
$s_1^n +> s_2^n$	Right-biased choice: first try to apply s_2 and if that fails then try to apply s_1.
$transient(s^n)$	A unary combinator restricting the application of s^n. (See discussion in Section 3.6).

Figure 6: The primitive constructs, combinators and constants of TL

TDL	A top-down left-to-right traversal. That is, parents are visited before children, and children are visited from left to right.
TDR	A top-down right-to-left traversal.
BUL	A bottom-up left-to-right traversal.
BUR	A bottom-up right-to-left traversal.
TDL_BR	A top-down left-to-right traversal where the strategy is broadcast (see discussion).

Figure 7: Some basic first-order traversals of TL

3.5 Higher-Order Strategies in TL

In TL a second-order strategy s^2 can be applied to a term t yielding a first-order strategy s^1. More generally, the application of a strategy of order n to a term t will result in a strategy of order $n-1$. The purpose of a second-order strategy is to create a first-order strategy containing data that is specific to a particular term. Typically this means that one or more schema variables will be bound to specific terms. For example, suppose that in the context of identifier renaming the identifier x is to be renamed to y. In this case, it would be convenient if a rule of the form $ident[[x]] \rightarrow ident[[y]]$ could be generated and applied to the appropriate terms. TL lifts and extends this idea to a higher-order framework.

For example, consider the abstract grammar shown in Figure 10.
Given this grammar, let us consider the second-order rewrite rule s^2 shown below:

$$s^2 : g[[i_1\ data_1]] \rightarrow (g[[i_2\ i_1]] \rightarrow g[[i_2\ data_1]])$$

Informally speaking, the rule s^2 can be seen as a template for relating information between the terms $data_1$ and i_1 in a specific context. The application $s^2(g[[1\ b]])$ yields the first-order strategy $g[[i_2\ 1]] \rightarrow g[[i_2\ b]]$, and the application $(g[[i_2\ 1]] \rightarrow g[[i_2\ b]])(g[[2\ 1]])$ yields $g[[2\ b]]$. In this instance, s^2 provides a vehicle for distributing data from $g[[1\ b]]$ to $g[[2\ 1]]$.

In general, a higher-order traversal traverses a term and applies a higher-order strategy s^n to every term encountered. Because the strategy being applied is of order n, the result of an application will

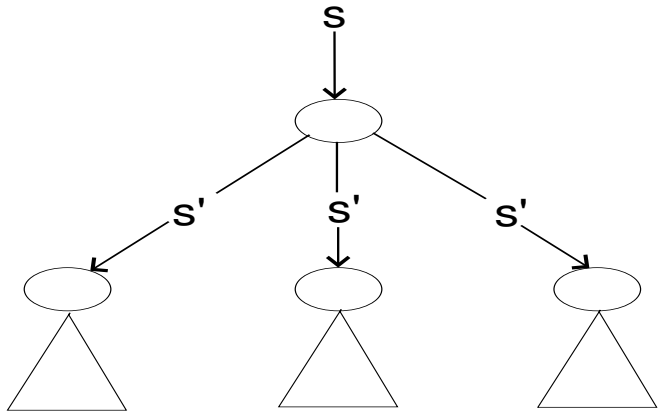

Figure 8: Diagram of a *tdl-broadcast* traversal from the perspective of strategy application

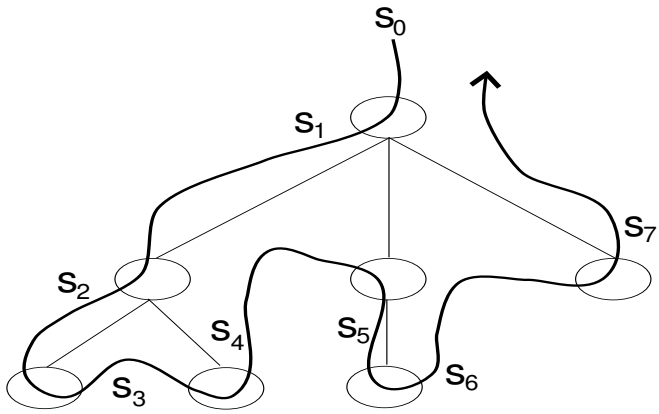

Figure 9: Diagram of a *tdl* traversal from the perspective of strategy application

be a strategy of order $n-1$. If a traversal visits m terms, then m strategies of order $n-1$ will be produced. Let s_1^{n-1}, s_2^{n-1} , ... , s_m^{n-1} denote the strategies resulting from such a traversal. Let $\oplus$ denote a binary combinator such as sequential composition, left-biased choice, or right-biased choice. In TL, binary strategic combinators can be used to combine strategic results into a single strategy. That is, higher-order traversals will combine a sequence of resultant strategies s_1^{n-1}, s_2^{n-1}, ... , s_m^{n-1} into a strategy of the form:

$$s_1^{n-1} \oplus s_2^{n-1} \oplus ... \oplus \ s_m^{n-1}$$

TL supports a number of higher order traversal including one called *seq_tdl* that traverses a term in a TDL fashion and composes the resulting strategies using the sequential composition operator. This higher order traversal is used in the method table construction example discussed in Section 4.3.

g	::=	i data
data	::=	i | char
i	::=	integer

Figure 10: A small abstract grammar

3.6 TL's *transient* Combinator

The *transient* combinator is a special combinator in TL. This combinator restricts a strategy so that it may be applied at most once. This restriction motivates the introduction of strategic constant *skip* into the framework of TL.

Operationally, we define a strategy of the form *transient*(s) as a strategy that *reduces* to the strategy *skip* if the application of the strategy s has been observed. Thus, transients open the door to *self-modifying* strategies. When using a traversal to apply a self-modifying strategy to a term, a different strategy may be applied to every term encountered during a traversal.

We now consider an example demonstrating the behavior of transients. Figure 11 shows a simple grammar defining a list of items, where an item may be either an identifier or a integer. Figure 12 shows a left-biased composition of three transient strategies. Each transient strategy will rewrite the identifier b to a different integer value. Let s denote the strategy in Figure 12 and let t denote the term *item_list*[[b b b]] shown in Figure 13. The evaluation of the strategic expression $TDL(s)t$ will yield the term *item_list*[[1 2 3]].

It is worth noting that the strategy s can only be successfully applied to terms of the form *item*[[b]], and there are three such terms in t. The strategy will fail to apply to any other term. During the course of a TDL traversal of t, the left-most occurrence of the identifier b is the first *item* encountered. The first transient in s will successfully apply to this item. This application will cause the transient to be reduced to *skip* while simultaneously rewriting b to the integer 1. As a result of the left-biased composition, no other strategies will be applied to this term. The next *item* that the traversal encounters is the second b. This time, the second transient (in the original strategy) will be applied to b. As a result of this rewrite, this transient will also be reduced to *skip*, and the second b will be rewritten to 2. The remaining occurrence of b is processed in a similar fashion.

In contrast, the evaluation of the strategic expression $TDL_BR(s)t$ will yield the result *item_list*[[1 1 1]]. The root of the parse tree is the nonterminal *item_list*. Its left child is the *item* corresponding to the first occurrence of b. Its second child is another *item_list*. The application of s to the root will fail. A copy of s will be given to both children of the root. This will cause the first child (an *item*) to be rewritten to 1 and the corresponding transient to reduce to *skip*. However, this strategy reduction will not effect the copy of s which is passed to the second child. This second child is again an *item_list* to which s is being applied. Since this second child is an *item_list* the same analysis holds. So again, the first child of this node is an *item* that will be rewritten to 1, and so on.

item_list	::=	item item_list | ϵ
item	::=	id | int
id	::=	identifier
int	::=	integer

Figure 11: A small grammar

$$transient(item[[b]] \rightarrow item[[1]]) <+ transient(item[[b]] \rightarrow item[[3]])$$

Figure 12: A left-biased composition of three transient strategies

4 A Class Loader for Java

At Sandia National Laboratories, a subset of the Java Virtual Machine (JVM) has been developed in hardware for use in high-consequence embedded applications. The implementation is called the *Sandia Secure Processor* (SSP) [16]. An application program for the SSP is called a *ROM image* and consists of a collection of class file-like structures stored on a read-only memory. The SSP is a closed system in the sense that all the structures used during execution must be present in the ROM image prior to execution. The closed nature of the SSP's execution environment enables the class loading activities of the JVM to be performed statically, prior to execution. Under these conditions, the functionality of the class loader is well-suited to a strategic implementation.

In the discussion that follows, we assume that an *application* consists of one or more Java *class files* and that Java class files have the structure defined in Lindholm and Yellin [15]. For the purposes of this discussion it is important to know that class files contain:

1. a *class* entry denoting the name of the class;
2. a *super* entry denoting the name of the superclass; and
3. a *methods section* containing all of the methods declared within the class.

4.1 Method Table Requirements for the SSP

When implementing a Java Virtual Machine (JVM), method tables are often used as a mechanism for indirectly providing access to the methods associated with an object [25]. Each class file has one method table whose entries contain information about particular methods such as the address of the first byte code of a method and the address of the constant pool corresponding to a method. The SSP has been designed in such a way that in order to provide correct information at runtime, it is sufficient for method tables within a class hierarchy to satisfy the properties given below.

Let s_m denote the signature of method m. Let (C, s_m) denote a method table entry corresponding to a method having a signature s_m that is defined in class C. Let T_C denote the method table for the class C, and let $\prec$ denote a reflexive, transitive sub-type relationship between classes as defined by the

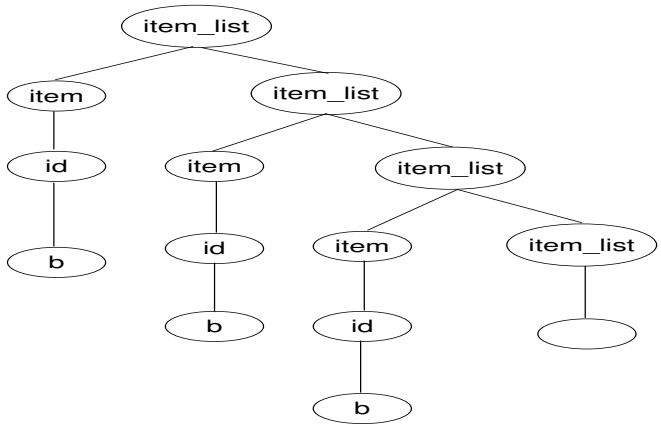

Figure 13: The structure of the term *item_list*[[*b b b*]]

Java *extends* directive. For example, given two classes B and C, if $C \prec B$ then either $C = B$ or C is a descendent of B within the inheritance hierarchy.

1. For every inherited method m that is **not** redefined in C there must be a corresponding entry of the form $(B, s_m) \in T_C$ where the class B denotes most recent ancestor of C where m is defined.

2. For every method m (re)defined in a class C there must be a corresponding entry $(C, s_m) \in T_C$.

3. The method table for the class C may only contain entries corresponding to inherited methods or methods that have been defined in C.

4. $\forall D_1, D_2,\ s_m \exists B, C, i, j : T_{D_1}[i] = (B, s_m) \wedge T_{D_2}[j] = (C, s_m) \wedge D_1 \prec D_2 \rightarrow i = j$. That is, within an inheritance hierarchy table entries corresponding to the signature s_m must reside at the same location (index) in all tables containing such an entry. For example, if information for the most recent definition of the method *foo* (e.g., C.foo) resides at the second entry of the method table for D_2, then all descendant classes of D_2 (e.g., D_1) must have the information for their most recent definition of *foo* (e.g., B.foo) as the second entry in their method tables.

The properties above permit method tables to be constructed in a concatenated fashion provided that entries associated with redefined methods destructively overwrite the corresponding inherited method table entries (e.g., the method table entry for the new definition of *foo* overwrites the method table entry for the old definition *foo*).

4.2 An Example of Method Table Construction

Figure 14 gives an abstraction of a Java application consisting of the classes A, B, C, and D. In the abstraction only the information relevant to method table construction is shown. Specifically, class A has *Object* as its super class and declares the methods $f1$, $f2$, and $f3$. Class B has class A as its

Class	Super	Methods
A	Object	f1() f2() f3()
B	A	f1() f4()
C	A	f2() f4()
D	C	f1() f4()

Figure 14: A simple application

super class and redeclares method $f1$ and declares method $f4$. The remaining classes C and D can be similarly described.

The method tables shown in Figure 15 satisfy the SSP's method table requirements given in Section 4.1.

Class	Method Table
A	info for A.f1() info for A.f2() info for A.f3()
B	info for B.f1() info for A.f2() info for A.f3() info for B.f4()
C	info for A.f1() info for C.f2() info for A.f3() info for C.f4()
D	info for D.f1() info for C.f2() info for A.f3() info for D.f4()

Figure 15: Abstract method table entries for the classes A, B, C, and D

4.3 Constructing SSP Method Tables in TL

The BNF grammar shown in Figure 16 is a greatly simplified description of the structure of a Java application. The grammar defines a Java application as consisting of an unordered list of Java class files. The class file structure has been simplified by abstracting away all structural elements that are irrelevant to the problem of method table construction. The grammar shown defines a *children* element as part of a class file. This element is not part of the class file structure as defined by the specification of the JVM [15] and has been added to facilitate strategic objectives.

app	::=	app cf | ϵ
cf	::=	"{" class super "[" methods "]" children "}"
class	::=	id
super	::=	id
methods	::=	mt "," method_list
mt	::=	mt_entry mt | ϵ
mt_entry	::=	key
method_list	::=	m_entry method_list | ϵ
m_entry	::=	key "(" ")"
children	::=	children cf | ϵ
key	::=	id "." id
index	::=	integer
id	::=	ident

Figure 16: A simplified grammar for class files

This discussion of method table construction begins at the point where the indexes within class files have been resolved to symbolic references by an earlier transformation phase. For example, the *class* and *super* elements for a class C are no longer constant pool indexes but are identifiers respectively corresponding to the symbolic references for name of C and the name of C's super class. We assume that method table entries consist of symbolic references containing the following information: (1) the class where the method is defined, (2) the name of the method, and (3) the method's descriptor.

Our strategic approach to solving the method table construction is as follows: First, class files are arranged into a tree structure reflecting their inheritance relationships. For example, if class B extends class A then class B will become a child of class A. The strategies for creating this inheritance tree are shown in Figure 17. Second, the resulting inheritance tree structure is processed in a top down fashion, inserting methods into method tables as we go. In particular, all the locally declared methods are destructively inserted into the method table of the current class as well as the method tables of every class that inherits from the current class. The strategies for accomplishing this are shown in Figure 21.

The execution of the *create_hierarchy* strategy shown in Figure 17 is described here. The evaluation of the expression ($seq_tdl\ subtype\ app_0$) traverses the application app_0 in a top-down left-to-right fashion and applies the higher-order strategy *subtype* to every term encountered. The results from this application are then sequentially composed into a strategy that is then applied by the first-order traversal TDL to the term $app[[\{obj\ obj[,]\}]]$. The term $app[[\{obj\ obj[,]\}]]$ denotes the *Object* class, which in this

example is assumed to contain no method declarations. In practice, the contents of the *Object* class is fixed so the *create_hierarchy* can be easily changed to describe its proper contents.

Let app_0 denote the application discussed in Section 4.2, and let cf_A, cf_B, cf_C, and cf_D denote the terms corresponding to the classes A, B, C, and D respectively. Under these conditions the evaluation of (*seq_tdl subtype* app_0) will create the sequentially composed transient strategies shown in Figure 18.

When applied to the term $app[[\{obj\ obj[,]\}]]$, the first transient in the this strategy inserts the term denoted by cf_A into the (empty) children list of obj. Then the transient combinator causes this strategy to be reduced to *skip*. Thus cf_A will not be inserted anywhere else in the hierarchy. The strategy that is returned from this application is shown in Figure 19. Similarly, when applied to the term cf_A the application of the strategy shown in Figure 19 causes the terms denoted by cf_B and cf_C to be inserted into the children list of cf_A at which time both corresponding transient strategies are reduced to *skip*. The strategy that is returned from this application is shown in Figure 20.

Finally, when applied to the term cf_C, the application of the strategy shown in Figure 20 causes the term denoted by cf_D to be inserted into the children list of cf_C at which time it also will be reduced to *skip*. At this point, the construction of the inheritance hierarchy is complete.

The strategy *construct_table* shown in Figure 21 is responsible for constructing method tables within a term structure corresponding to a class hierarchy constructed by the *create_hierarchy* strategy. The evaluation of the strategic expression $TDL(distribute_entries)app_0$ causes the *distribute_entries* strategy to be applied to parent classes before children classes. In particular, the strategy application will be consistent with the subtype ordering of the inheritance hierarchy. Taking a closer look at the *distribute_entries* strategy, the evaluation of the strategic expression (*seq_tdl merge_methods* $method_list_1$) traverses the method list of the class cf_0 to which the strategy is being applied. For each method encountered it will produce a strategy of the form:

$$transient(mt[[id_3\ .\ id_2\ mt_2]] \rightarrow mt[[id_1\ .\ id_2\ mt_2\ ]] <+\ mt[[\ ]] \rightarrow mt[[id_1\ .\ id_2]])$$

This strategy is capable of destructively overwriting a method table entry containing a prior definition of the same method or of appending a method to the end of a method table. The application must still ensure that within an inheritance hierarchy, the method table entries corresponding to parent class method definitions are inserted into the method tables of all descendants before insertions of descendant classes are performed. This is accomplished by a *TDL_BR* traversal.

Returning to the simple application discussed in Section 4.2, we assume that the class files for *obj*, A, B, C, and D have been formed into an inheritance hierarchy. In the strategy *construct_tables*, the evaluation of the strategic expression $TDL(distribute_entries)app_0$ visits the classes in the following order: *obj*, A, B, C, and D. To each of these classes, the strategy *distribute_entries* is applied.

Since *obj* has no methods and inherits no methods, its method table is empty. So without loss of generality our discussion begins with the term corresponding to the class A. Let $method_list_1$ denote the method list for A. The evaluation of the strategic expression (*seq_tdl merge_methods* $method_list_1$) generates the strategy shown below for the methods declared in A.

$$transient(mt[[id_2\ .\ f1()\ mt_2]] \rightarrow mt[[A\ .\ f1()\ mt_2\ ]] <+\ mt[[\ ]] \rightarrow mt[[A\ .\ f1()]])$$
$$transient(mt[[id_2\ .\ f2()\ mt_2]] \rightarrow mt[[A\ .\ f2()\ mt_2\ ]] <+\ mt[[\ ]] \rightarrow mt[[A\ .\ f2()]])$$
$$transient(mt[[id_2\ .\ f3()\ mt_2]] \rightarrow mt[[A\ .\ f3()\ mt_2\ ]] <+\ mt[[\ ]] \rightarrow mt[[A\ .\ f3()]])$$

create_hierarchy	:	$app_0 \rightarrow TDL(seq_tdl\ subtype\ app_0)\ app[[\{obj\ obj[,]\}]]$
subtype	:	$cf_0 \rightarrow transient(add_subtype(id_2,cf_0))$
	if	$cf_0 \gg cf[[\{\ id_1\ id_2\ methods_1\ children_1\}]]$
$add_subtype(id_2, cf_0)$	:	$cf[[\{id_2\ id_3\ methods_2\ children_2\}]] \rightarrow cf[[\{id_2\ id_3\ method_2\ children_2\ cf_0\}]]$

Figure 17: Strategies for creating a class hierarchy

$$
\begin{array}{lcl}
transient(cf[[\{obj\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_A\}]])\ ; \\
transient(cf[[\{A\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_B\}]])\ ; \\
transient(cf[[\{A\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_C\}]])\ ; \\
transient(cf[[\{C\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_D\}]])\ ;
\end{array}
$$

Figure 18: The sequential composition of transient strategies

$$
\begin{array}{lcl}
skip\ ; & & \\
transient(cf[[\{A\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_B\}]])\ ; \\
transient(cf[[\{A\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_C\}]])\ ; \\
transient(cf[[\{C\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_D\}]])\ ;
\end{array}
$$

Figure 19: The strategy remaining after inserting cf_A into the children list of *obj*.

$$
\begin{array}{lcl}
skip\ ; & & \\
skip\ ; & & \\
skip\ ; & & \\
transient(cf[[\{C\ id_3\ methods_2\ children_2\}]] & \rightarrow & cf[[\{obj\ id_3\ method_2\ children_2\ cf_D\}]])\ ;
\end{array}
$$

Figure 20: The strategy remaining after inserting cf_B and cf_C into the children list of cf_A.

construct_tables	:	$app_0 \rightarrow TDL(distribute_entries)app_0$
distribute_entries	:	$cf_0 \rightarrow TDL_BR\ (\ seq_tdl\ merge_methods\ method_list_1\ )\ cf_0$
	if	$cf_0 \ll cf[[\ \{class_1\ super_1\ [mt_1,method_list_1]\ children_1\}\ ]]$
merge_methods	:	$m_entry[[id_1\ .\ id_2\ ()\ ]] \rightarrow transient(merge(id_1,id_2))$
$merge(\mathrm{id}_0,\mathrm{id}_1)$	:	$mt[[id_3\ .\ id_2\ mt_2]] \rightarrow mt[[id_1\ .\ id_2\ mt_2\ ]] <\!\!+\ mt[[\ ]] \rightarrow mt[[id_1\ .\ id_2\ ]]$

Figure 21: Strategies for inserting methods into method tables

This strategy is applied using the broadcasting traversal TDL_BR, which accomplishes the following: (1) the method table for A is populated, and (2) the method table for every descendant of A is populated with entries corresponding to $A.f1()$, $A.f2()$, and $A.f3()$. At this point, the method table for A is complete, and the method tables for all the descendants of A are not yet complete. They are identical to the method table of A. The class B is the next class to which the *distribute_entries* strategy is applied. The strategy resulting from this application is shown below:

$$transient(mt[[id_2\ .\ f1()\ mt_2]] \rightarrow mt[[B\ .\ f1()\ mt_2\ ]] <\!\!+\ mt[[\ ]] \rightarrow mt[[B\ .\ f1()]])$$
$$transient(mt[[id_2\ .\ f4()\ mt_2]] \rightarrow mt[[B\ .\ f4()\ mt_2\ ]] <\!\!+\ mt[[\ ]] \rightarrow mt[[B\ .\ f4()]])$$

The TDL_BR traversal causes this strategy to be applied to the method table of B as well as the descendants of B (of which there are none). The application overwrites the entry for $A.f1()$ to $B.f1()$ and adds an entry for $B.f4()$ to the end of B's method table. At this point the method table for B is complete.

The next class to which the *distribute_entries* strategy will be applied is C. The strategy resulting from this application is shown below:

$$transient(mt[[id_2\ .\ f2()\ mt_2]] \rightarrow mt[[C\ .\ f2()\ mt_2\ ]] <\!\!+\ mt[[\ ]] \rightarrow mt[[C\ .\ f2()]])$$
$$transient(mt[[id_2\ .\ f4()\ mt_2]] \rightarrow mt[[C\ .\ f4()\ mt_2\ ]] <\!\!+\ mt[[\ ]] \rightarrow mt[[C\ .\ f4()]])$$

The TDL_BR traversal causes this strategy to be applied to the method table of C as well as the descendants of C, namely the class D. The application overwrites the entry for $A.f1()$ to $C.f1()$ and adds an entry for $C.f4()$ to the end of the method tables for both C and D. At this point the method table for C is complete and the traversal proceeds on to D, which is processed in a similar fashion.

5 Assurance

Sandia requires the development of the SSP to produce strong evidence of the correctness of the system. The class loader is a weak link in the assurance chain of the SSP. Commercial compilers take Java source code and produce class files. The assurance provided by a commercial compiler stems from several sources including the fact that the Java community at large performs an extensive stress test of the compiler. Over time, such a testing environment causes a software product to mature. While bugs may still exist in the compiler, the likelihood of encountering a bug in the class loader is significantly greater. Thus, considerable effort is being devoted towards providing assurance in the translation performed by the SSP class loader. In particular we are interested in applying formal reasoning techniques to verify general properties of the class loader.

The assurance provided by formal reasoning comes in the form of a mathematical proof and typically involves a model of the system under analysis (rather than the actual system itself). General properties are stated in terms of theorems involving the model. The proof of theorems provides strong assurance that the model behaves as required. Assurance of the correctness of the system under analysis relies on (1) confidence that the proofs themselves are sound and (2) confidence that the model faithfully describes the system: theorems that hold for the model hold for the system. While it is theoretically possible to automate the construction of proofs, in practice it is extremely difficult and requires sophisticated tools and approaches.

We are using the modeling and verification framework provided by ACL2 [11][12] to formally prove theorems about the class loader. ACL2 is a programming language based on the applicative subset of Common Lisp. In this language, users can build executable models of software systems. ACL2 is also a tool that assists users in proving theorems about their ACL2 programs.

The class loader is modeled as a system that consists of states and state transisitons. The state is modeled by the class files of the Java application, and the state transitions are defined by the transformations on this application. The JVM and the SSP provide the basis for formally understanding equivalence between states.

A model of HATS is constructed by defining an abstract machine that controls the application of transformation rules. Each transformation rule modifies the state of the system. The abstract machine operates according to the following sequence: fetch the next transformation rule and node from the current state, apply the transformation, and return a new machine state.

Though we are exploring the verification of a number of properties of the class loader, our ultimate goal is to verify that the transformation rules preserve the meaning of the term to which they are applied (i.e., the class loader is correct). In the context of the SSP, the initial term is a set of class files, C_0, generated by a Java compiler. The semantics of this term is defined by the JVM specification. We can think of the JVM as defining a mapping from $(classfiles \times inputs)$ to $outputs$. Let $Eval_{JVM} : classfiles * inputs \rightarrow outputs$ denote this mapping function. $Eval_{JVM}$ defines the behavior of the program encoded in the class files. The final term is a ROM image, which we denote C_{ROM}. The semantics of this term is defined by the SSP hardware, $Eval_{SSP}$. HATS accomplishes the conversion of C_0 to C_{ROM}, as indicated by the notation $C_{ROM} = T^*(C_0)$. In this notational framework, what must be shown for inputs I is:

$$\forall (C_0, I) Eval_{JVM}(C_0, I) = Eval_{SSP}(T^*(C_0), I)$$

The problem above can be decomposed by defining a sequence of normal forms, $C_0, C_1, C_2, ...$ in the transformation of C_0 to C_{ROM}. Properties about these normal forms are formally specified and become theorems within our verification framework. Constant pool normalization and field distribution are two examples of transformations leading to normal forms. In constant pool normalization all indirection is removed from the constant pool entries of the class files in C_0. Let T^1 denote the transformation that accomplishes this task. Similarly, let T^2 denote the normal form resulting from field distribution. At present, a sequence of five intermediate normal forms have been defined. For each normal form, there is an evaluation function, $Eval_n$. Thus, the original correctness conjecture can be restated as a sequence of conjectures:

$$\begin{aligned}
&\forall (C_0, I) Eval_{JVM}(C_0, I) = Eval1(T^1(C_0), I) \\
&= Eval_2(T^2(T^1(C_0)), I) \\
&\\
&\ldots \\
&= Eval_{SSP}(T^*(C_0), I)
\end{aligned}$$

where T^* is the composition of the individual transformations. This allows the proof to be constructed incrementally, and therefore, reduces the complexity of the proof.

6 Related Work

ASF+SDF [5], ELAN [3], Stratego [27] and Maude [8] are operational systems that can be used for program transformation. The ρ-calculus [7] and the S'_γ-calculus [13] are theoretical frameworks in which program transformation can be considered. With the exception of the ρ-calculus, none of these systems directly support higher-order rewrite rules or strategies. Furthermore, none of these systems support the dynamic (i.e., at runtime) construction of strategies or the *transient* combinator.

However, Stratego does support the ability to dynamically add and remove rules to an existing rule base [26]. This capability can be understood as a restriction of the dynamic strategy creation possible in TL. Stratego also supports the ability to construct rules having contextual matches [28]. Contextual matching enables non-local information to be brought together within a match expression. It turns out that the behavior of a contextual match can be directly implemented using the *match* and *build* primitives of Stratego within the context of a nested traversal. Contextual matches can also be readily simulated using the higher-order capabilities of TL.

ELAN [3] is a first-order failure-based rewrite system in which an AC matching algorithm [9] can be used as the mechanism for the syntactic comparison of terms. ELAN is a strategic system whose semantic foundation rests upon the ρ-calculus. Rewrite rules can be labeled and one or more rules may share the same label. Thus labels are bound to rule bases. The consequence of AC matching and labeled rule bases is that the application of a rule (base) to a specific term may yield multiple results. This form of non-determinism surrounding rule base application is central to ELAN and gives the system a deductive/declarative flavor. ELAN provides a variety of choice combinators together with a backtracking capability as mechanisms for dealing with the non-determinism.

ASF+SDF [2] is a first-order identity-based rewriting framework in which an extended form of matching provides the mechanism for the syntactic comparison of terms. The extension to matching permits associative matching on lists structures. In [5] ASD+SDF is further developed so that one can combine parameterized rewrite rules with a fixed set of generic traversals. The result of such a combination is a *traversal function* – which is essentially a rewrite rule annotated with an appropriated predefined traversal. One of the goals in [5] is to provide primitives so that the resulting traversal functions can be used in a type-safe manner.

ACL2 has been used to prove the correctness of hardware implementations of microprocessors and floating point algorithms as well as parts of implementations of the JVM. Our approach to modeling the class loader is based on a heavily-researched model in which a system is described in terms of states and state transitions.

Boyer and Yu used Nqthm, the predecessor of ACL2, to formalize a substantial subset of a commercial microprocessor, the Motorola MC68020[4]. Based on this model, they were able to verify many binary machine code programs produced by commercial compilers from source code in such high-level languages as Ada, Lisp, and C.

Moore[17][21] also used the same approach to model Piton, an assembly programming language that is implemented on a microprocessor, the FM8502, via a compiler, an assembler, and a linker. A

piton interpreter was coded in the ACL2 logic in which given an initial state p0 you obtain state pn by running piton forward n steps. However, the alternative approach is to map p0 to down to a FM8502 state (or core image), run the FM8502, and map the resulting state back up. The compiler, assembler and linker were also defined as functions in the ACL2 logic. The implementation of was mechanically proved correct.

More recently, this type of model has been used to reason about the behavior of computations described in terms of Java byte codes[18][19][20][22].

The general approach was to model a significant subset of the JVM operationally using ACL2. This model was used to execute certain Java programs by compiling them into bytecode. The model consists of a state of the JVM and state transition function for each JVM bytecode instruction in the subset. Basically, the state is a triple containing a thread table, a heap, and a class table. The transition function takes an instruction, a thread, and a state, and returns a new state that is the result of executing the given instruction on the given thread in the given state. The new state is a modification of the previous state.

7 Conclusion

In this article we have argued that higher-order rewrite rules and strategies enhance the ability of rewriting techniques to concisely realize complex transformational objectives, specifically objectives surrounding the manipulation of non-localized data. The implications for software development are that (1) the techniques of higher-order rewriting may enable more complex transformational objectives to be realized in a practical setting, and (2) the conciseness offered by higher-order rules and strategies could positively impact the ability of automated reasoning systems like ACL2 to verify the correctness of transformations.

TL is a higher-order strategic programming system with several unique features including: (1) the ability to dynamically construct strategies, (2) the ability to control strategy application through a special combinator called a *transient* thereby opening the door to self-modifying strategies, and (3) the ability to broadcast a copy of a strategy to every child of a term. When combined, these capabilities have a synergistic effect enabling a variety of transformational objectives to be solved in an elegant fashion. The effectiveness of this paradigm is shown by demonstrating how method tables can be constructed for abstract Java class files.

The implementation of the JVM being targeted by this demonstration is the Sandia Secure Processor (SSP). The SSP is a hardware implementation of a subset of the JVM for use in high-consequence safety-critical embedded systems. Within the SSP, method tables are used as a mechanism for indirectly providing access to the methods associated with an object. The strategic programming concepts shown have been successfully scaled to the real world, realizing the entire class loader for the SSP. This implementation was accomplished in HATS, a restricted dialect of TL that is freely available [10].

The Sandia Secure Processor project places a high priority on the provision of strong evidence that the system behaves correctly. Currently, a weak link in the chain of evidence is in the software that performs class loading. In order to provide strong evidence of correctness in the strategic programming solution to the class loader problem, we are developing a framework for proving correctness. This framework, based on the ACL2 theorem proving system, is intended to show that the transformation of

terms preserves the initial semantics. A difficulty encountered in such a verification is how to formally state the notion of functional equivalence between the different incarnations of class files that arise during the course of transformation. This problem is solved by a series of $Eval_i$ functions, with each function being capable of processing class files at a particular levels of abstraction (e.g., prior to symbolic resolution). When completed, the nature of the evidence provided by such a formal verification of the class loader can constitute a major component of an argument that the class loader for the SSP meets its dependability requirements.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and all That.* Cambridge University Press, 1998.

[2] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[3] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. *An Overview of ELAN.* In C. Kirchner and H. Kirchner, eds., International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science, France, 1998. Elsevier Science.

[4] Robert S. Boyer and Yuan Yu. *Automated proofs of object code for a widely used microprocessor.* Journal of the ACM, 43(1):166-192, January 1996.

[5] M.G.J. van den Brand, P. Klint, and J.J. Vinju. *Term Rewriting with Traversal Functions.* ACM Transactions on Software Engineering and Methodology (TOSEM), 12:2, pp 152-190, 2003.

[6] M. van den Brand, A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.* IWPC 1998, June 24-26, Ischia, Italy.

[7] H. Cirstea and C. Kirchner. *Intoduction to the rewriting calculus.* INRIA Research Report RR-3818, December 1999.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí

[9] S. Eker. *Associative-commutative matching via bipartite graph matching.* Computer Journal, 38(5):381-399, 1995.

[10] *http://faculty.ist.unomaha.edu/winter/hats-uno/HATSWEB/index.html*

[11] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[12] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: Case Studies.* Kluwer Academic Publishers, June 2000.

[13] R. Lämmel. *Typed Generic Traversal With Term Rewriting Strategies.* Journal of Logic and Algebraic Programming, Vol 54, pp 1–64, 2003.

[14] R. Lämmel, E. Visser, and J. Visser. *The Essence of Strategic Programming*. Draft.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification 2nd Edition*. Addison-Wesley, Reading, Massachusetts, 1999.

[16] J. A. McCoy. *An Embedded System For Safe, Secure And Reliable Execution Of High Consequence Software*. Proceedings of the 5th IEEE International High-Assurance Systems Engineering Symposium, Nov. 2000.

[17] J S. Moore. *Piton – A Mechanically Verified Assemply-Language*. Kluwer Academic Publishers, 1996.

[18] J S. Moore. *Proving Theorem about Java-like byte code*. In E.-R. Olderog and B. Steffen, editor, correct system design-Recent Insight and Advances, pages 139-162, Heidelberg, 1999. LNCS 1710

[19] J S. Moore. *Proving Theorems about Java and the JVM with ACL2*. Models, Algebras and Logic of Engineering Software, M. Broy and M. Pizka (eds), IOS Press, Amsterdam, pp 227-290, 2003.

[20] J S. Moore and Robert S. Boyer. *Mechanized Formal Reasoning about Programs and Computing Machines*. In R. Veroff (ed.), Automated Reasoning and Its Applications: Essays in Honor of Larry Wos , MIT Press, 1996.

[21] J S. Moore, Tom Lynch, and Matt Kaufmann. *A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm*. IEEE Transactions on Computers, 47(9), pp. 913-926, Sep., 1998.

[22] J S. Moore. *Proving theorems about Javalike byte code*. In E.-R. Olderog and B. Steffen, Eds., Correct System Design-Recent Insights and Advances, LNCS 1710, pp. 139-162. Springer-Verlag, Berlin 1999.

[23] H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer-Verlag 1990.

[24] S. Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall 1993.

[25] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1998.

[26] 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.

[27] E. Visser. *Language Independent Traversals for Program Transformation*. In Johan Jeuring, editor, Workshop on Generic Programming (WGP'00), Ponte de Lima, Portugal, July 2000.

[28] E. Visser. *Strategic Pattern Matching*. In: Rewriting Techniques and Applications (RTA '99), Trento, Lecture Notes in Computer Science (1999).

[29] V.L. Winter and M. Subramaniam. *The Transient Combinator, Higher-Order Strategies, and the Distributed Data Problem.* Science of Computer Programming (accepted).

[30] V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software.* Advances in Computers, vol. 58, Academic Press, Amsterdam, pp 49-117, 2003.

Distribution:

1	MS0510	Greg Wickstrom, 2116
1	MS0510	James McCoy, 2116
1	MS0510	Anna Schauer, 2116
2		Victor Winter PKI 174 C 1110 South 67th Street Omaha, NE 68182
2		Steve Roach Department of Computer Science University of Texas at El Paso El Paso, TX 79968-0518
1	MS9018	Central Technical Files, 8945-1
2	MS0899	Technical Library, 9616