SANDIA REPORT

SAND2004-3225
Unlimited Release
Printed August 2004

An Abstract Class Loader for the SSP and its Implementation in TL

Greg Wickstrom
Surety Electronics and Software
Department 2116
PO Box 5800
Albuquerque, NM 87185-0510

Victor Winter and Jason Beranek
University of Nebraska at Omaha
Omaha, Nebraska 68182

Steve Roach and Fares Fraij
University of Texas at El Paso
Department of Computer Science
El Paso, TX 79968

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

SAND2004-3225
Unlimited Release
Printed August 2004

An Abstract Class Loader for the SSP and its Implementation in TL

Greg Wickstrom
Surety Electronics and Software Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Victor Winter and Jason Beranek
Sandia Contract No. 5137
University of Nebraska at Omaha
Department of Computer Science
Omaha, Nebraska 68182

Steve Roach and Fares Fraij
University of Texas at El Paso
Department of Computer Science
El Paso, Texas 79968

Abstract

The SSP is a hardware implementation of a subset of the JVM for use in high consequence embedded applications. In this context, a majority of the activities belonging to class loading, as it is defined in the specification of the JVM, can be performed statically. Static class loading has the net result of dramatically simplifying the design of the SSP as well as increasing its performance. Due to the high consequence nature of its applications, strong evidence must be provided that all aspects of the SSP have been implemented correctly. This includes the class loader. This article explores the possibility of formally verifying a class loader for the SSP implemented in the strategic programming language TL. Specifically, an implementation of the core activities of an abstract class loader is presented and its verification in ACL2 is considered.

Contents

1 Introduction **7**
1.1 The ROM Images Executed by the SSP . . . 7
1.2 An Overview of Class Loader Requirements . . . 9

2 Overview of the Structure of Class Files **12**
2.1 Index Resolution . . . 12
2.2 Static Field Address Calculation . . . 14
2.3 Instance Field Offset Calculation . . . 15
2.4 Method Table Construction . . . 15
2.5 Inter-class Absolute Address and Offset Address Distribution . . . 17

3 An overview of TL **18**
3.1 The Basic Constructs of TL . . . 19
3.2 Tree/Term Notation . . . 20
3.3 Conditional Rewrite Rules . . . 21
3.3.1 Conditions . . . 21
3.3.2 Rule Application . . . 22
3.4 Combinators . . . 22
3.5 Generic First-Order Traversals . . . 23
3.6 Higher-Order Rules and Strategies . . . 23
3.7 Higher-Order Generic Traversals . . . 23

4 A Strategic Implementation of the Class Loader Core **24**
4.1 Index Resolution in TL . . . 28
4.2 Static Field Address Calculation in TL . . . 29
4.3 Instance Field Offset Calculation in TL . . . 30
4.4 Method Table Construction in TL . . . 32
4.5 Inter-class Absolute Address and Offset Address Distribution in TL . . . 34

5 Verification and Validation **35**
5.1 The ACL2 Theorem Prover . . . 35
5.2 Modelling TL in ACL2 . . . 36
5.3 Verification of the Correctness of the Transformation Rules . . . 37

6 Related Work **41**
6.1 Rewriting and Strategic Programming . . . 41
6.2 Verification . . . 42

7 Conclusion **42**

List of Figures

1 Resolution of indexes (aka. encoded symbolic references) 9
2 The components of a Java class file 13
3 The subset of the Java class file structure relevant to the class loader core 13
4 A constant pool description of the integer field B.x 14
5 Classes and their static fields 15
6 Mapping static fields to absolute heap addresses 15
7 Classes and their instance fields 16
8 Mapping instance fields to object offsets 16
9 Mapping instance fields to object offsets 16
10 Classes and their virtual methods 17
11 Method table construction 17
12 An examples of two classes having external symbolic references to fields and methods . 18
13 Inter-class distribution of field offset/address and method table offsets 18
14 A concrete syntax fragment 21
15 The syntactic structure of rewrite rules in TL 21
16 An extended-BNF grammar describing a simplified application in terms of a list of class files 25
17 Three abstract class files prior to class loading 26
18 The three abstract class files shown in Figure 17 after class loading 27
19 Index Resolution 28
20 A TL strategy for absolution address calculation 30
21 Offset address calculation 31
22 Method table construction 33
23 Inter-class distribution 35

Nomenclature

BNF	Backus-Naur Form
HATS	High Assurance Transformation System
IDE	Integrated Development Environment
JVM	Java Virtual Machine
ROM	Read Only Memory
SSP	Sandia Secure Processor
TL	Transformation Langauge – a higher-order strategic programming language

1 Introduction

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) [11][24] and supports a subset of Java bytecodes as its native instruction set.

This paper has three objectives: (1) to informally describe the core functionality of the class loader for the SSP, (2) to demonstrate how the abstract functionality of this core class loader can be implemented using higher-order strategic programming techniques, and (3) to discuss how the correctness of the class loader can formally verified. The paper is organized as follows. We begin with an introduction to the goals of the SSP class loader. Section 2 gives an overview of the Java class file structure and restricts our attention to the subset of class files impacted by the SSP class loader core. This section also gives an informal description of the core activities of the SSP class loader. Section 3 gives an overview of the higher-order strategic programming language TL. Section 4 presents and discusses a TL implementation of an abstract class loader core. Section 5 describes preliminary efforts at verifying and validating the transformations.

1.1 The ROM Images Executed by the SSP

An application program for the SSP is called a *ROM image* and consists of a collection of class file images stored on a Read-Only Memory (ROM). The image of a class file in the ROM contains (1) a ROM constant pool and (2) a method table and methods section. For the purposes of this article, there are two major differences between ROM constant pools and the constant pools found in Java class files. The first major difference is that in ROM constant pools, *class*, *field*, and *method* entries are represented in terms of absolute addresses or offset addresses together with additional *data* that, broadly speaking, provides type information describing the entry. In contrast, in Java class files such entries are represented as *encodings of symbolic references*. In this article, we refer to the aggregation of one or more constant pool indexes as an *encoding of a symbolic reference*. The second major difference is that, unlike the constant pools in Java class files, ROM constant pools do not contain any *name_and_type*, *name*, and *Utf8* entries. Thus constant pools in the ROM are limited to the following entries: *constant integer*, *constant long*, *static field*, *instance field*, *class*, and *method*. Among the method entries, in this article, we restrict our consideration only to virtual methods. That is, methods that are invoked within an application program using the invokevirtual bytecode. In this article, we do not consider static or special methods whose invocations are respectively achieved through the *invokestatic* and *invokespecial* bytecodes. This restriction to virtual methods is essentially without loss of generality. The one exception being an anomalous case involving dynamic binding of a method that has been invoked using the *invokespecial* bytecode [10][19].

In a ROM image, the methods section of class files have also been modified. In particular, within the bytecode of a method, constant pool indexes (i.e., encodings of symbolic references) have been replaced with offset addresses into the ROM constant pool. These are the major differences between class files as produced by a Java compiler and as they appear on the ROM.

Due to the similarity between the JVM and the SSP, the primary goal of the class loader for the SSP can be simply stated:

The Primary Goal of the SSP Class Loader:

Resolve *encoded symbolic references (indexes) in class files to absolute addresses or offset addresses.*

Though the focus on this paper is on the primary goal as stated above, we would like to mention that there are numerous secondary goals a class loader for the SSP must satisfy. These secondary goals include the construction of various kinds of type data associated with constant pool entries, padding method bodies so methods begin and end on word boundaries, and so on.

Within the SSP, absolute addresses belong to one of two domains. These domains model the address spaces of the ROM and heap respectively. The first domain, D_{ROM}, describes the location of bytes within the ROM image. Addresses belonging to D_{ROM} are used to describe the location of immutable structures associated with a class such as the location of a ROM constant pool, the starting address of a method, and the method table associated with a class. The second domain, D_{HEAP}, describes the location of bytes within the SSP's heap memory. From the perspective of the class loader, absolute addresses belonging to D_{HEAP} are used solely to describe the location of static fields. All other assignments of absolute addresses in D_{HEAP} occur during runtime. For example, during execution bytecodes such as *new* are responsible for heap memory allocation. These allocations yield absolute addresses in D_{HEAP} belonging to objects (i.e., object references).

In contrast to absolute addresses, offset addresses are used to describe the location of data in a relative fashion. Adding an offset address to an appropriate absolute address (such as the start of a constant pool) yields an absolute address that describes the location of a particular data item within an absolute address space (e.g., the address of a constant pool entry). The design of the SSP makes use of three types of offset addresses: *constant pool offsets*, *method table offsets*, and *object offsets*. Constant pool offset addresses are values belonging to the domain D_{CP} and describe offsets relative to the start of a ROM constant pool. Semantically speaking, constant pool offset addresses are used to describe the location of ROM constant pool entries. Method table offset addresses are values belonging to the domain D_{MT} and describe offsets relative to the start of a method table residing in the ROM. Semantically speaking, method table offset addresses are used to describe the location of entries in a method table. Method table entries contain information needed to execute a particular virtual method. This information includes the starting address of the virtual method, the absolute address of the constant pool associated with this virtual method, and type information defining the number of local variables and parameters used by the method. And lastly, object offset addresses are values belonging to D_{OBJ} and describe offsets relative to the start of an object (i.e., an instance of a class). Semantically speaking, object offset addresses are used to describe the location of data within an object. This data consists primarily of the object's instance fields. However, the first data entry of an object is an absolute address belonging to D_{ROM} describing the location of the class from which this object is an instance (this is needed when a method for this object is invoked).

Context where Index Occurs	Interpretation of Resolution
within a bytecode in the body of a method	ROM constant pool offset address where the type of the constant pool entry can be inferred from the bytecode
this_class	absolute address in ROM
super_class	absolute address in ROM
class entry in ROM constant pool	absolute address in ROM
static field entry in ROM constant pool	absolute address in heap
static field entry in fields section of class file	absolute address in heap
instance field entry in ROM constant pool	object offset address
instance field entry in fields section of class file	object offset address
virtual method entry in ROM constant pool	ROM method table offset address
virtual method entry in ROM method table	absolute address of method in ROM

Figure 1: Resolution of indexes (aka. encoded symbolic references)

Given an index, the type of address to which it should be resolved can be uniquely determined from the context in which it is used. Figure 1 lists the contexts that must be considered.

1.2 An Overview of Class Loader Requirements

Within an application program, classes may be ordered to form inheritance hierarchies through the Java "extends" directive. The subtype relation resulting from inheritance hierarchies impose constraints on how object offsets and method table offsets must be calculated. These constraints as well as others can be expressed in the form of properties that a resolved collection of encoded symbolic references must possess in order to be correct. A non-exhaustive list of these properties follows.

- **Unique-Offset**: Within the scope of an inheritance hierarchy, all instance fields must be resolved to unique object offsets. *Rationale:* This property ensures that two instance fields are not mapped to the same offset.

- **Consistent-Offset**: Instance fields must be resolved to offsets in a manner that is consistent with upcasting. *Rationale:* This property ensures that the mapping of instance fields to offsets is treated in a consistent fashion for all classes belonging to an inheritance hierarchy.

- **Non-Overlapping-Addresses**: The values of primitive types supported by the SSP must be mapped to memory regions that are sufficiently large to hold all legal values of that type. For example, an integer field must be mapped to a memory region that is at least 32-bits wide. *Rationale:* This property prevents instance fields from being packed so tightly within an object that their memory spaces overlap. This property also prevents static fields from being packed so tightly in the heap that their memory spaces overlap. Note that simply requiring that instance fields have unique offsets is not sufficient to ensure this property.

- **Unique-Address**: Within an application, all static fields must be resolved to a unique absolute address within the heap. *Rationale:* This requirement ensures that two static fields are not mapped to the same absolute address within the heap.

- **Consistent-Method-Invocation**: Virtual method invocations must be referred to indirectly via an offset to a method table. Furthermore, within an inheritance hierarchy all method tables must be consistent with respect to the positioning of method table entries. *Rationale:* Positional table entry consistency among method tables within an inheritance hierarchy ensures that proper method invocation will result in the presence of upcasting.

- **Address Units**

  - Constant pool offset addresses must be in word units (i.e., 32-bit quantities) in the range $0 \leq cp_offset < 2^{16}$. The only range exception being that of the *ldc* bytecode which has the range $0 \leq cp_offset < 2^{8}$. *Rationale:* This requirement ensures that spacial requirements of constant pool indexes in bytecodes match the spatial requirements of offset addresses into a ROM constant pool. This requirement also guarantees that the addressing scheme of the SSP will be able to handle all Java constant pools having 2^{15} elements or less, with 2^{15} being a conservative (guaranteed achievable) lower bound. The class loader must flag as an error all constant pools whose element count exceed this threshold.
  - Method table offsets must be in word units in the range $0 \leq mt_offset < 2^{16}$. *Rationale:* This design decision enables the addressing scheme of the SSP to handle Java classes whose total number of virtual methods (both declared and inherited) is less than or equal to 2^{16}. The class loader must flag as an error all classes whose virtual method count exceeds this threshold.
  - Absolute addresses in the heap must be in byte units in the range $0 \leq address < 2^{24}$. *Rationale:* This enables a Java constant pool entry containing a symbolic reference to a static field to be resolved to a one-word ROM constant pool entry describing the type (8-bits) and absolute heap address (24-bits) of the static field.

The list given above is not meant to be complete, but rather to give the reader a feeling for the necessary kinds of properties that must be satisfied in order for the resolution performed by a class loader to be considered correct. Taking a more rigorous approach, correctness properties of the kind just described can be formalized and expressed in terms of a formula $\mathcal{C}$ in first-order logic. Abstractly speaking, this formula defines properties and relationships between (1) encoded symbolic references and addresses, and (2) addresses within an address space.

In addition to correctness properties, the resolution of encoded symbolic references must also satisfy a number of efficiency-based constraints. A number of these constraints are related to optimization of memory usage and access. However, fault-tolerance, safety, and security constraints are also possible. A non-exhaustive list of these constraints follows.

- **Spatial-Efficiency**:

- Instance fields should be packed as tightly as possible in objects and static fields should be tightly packed in the heap. *Rationale:* This is a necessary condition to assure that heap memory is efficiently utilized.
- Object offset addresses must be in byte units (i.e., 8-bit quantities). *Rationale:* This design decision enables sequences of short, char, byte, and boolean fields to be closely packed within an object.

- **Temporal-Efficiency**:
  - 64-bit values (i.e., longs) should not span 32-bit boundaries within the heap or ROM address space. For example, a long may be stored at byte address 0x0000 or 0x0004 but should not be stored at byte address 0x0002 because retrieving a long value stored in this manner would require 3 memory fetches instead of 2. *Rationale:* This is a necessary condition to assure that the SSP executes efficiently.
  - 32-bit values such as integers and references should not span 32-bit boundaries within the heap or ROM address space. For example, an integer value may be stored at byte address 0x0000 or 0x0004 but should not be stored at byte address 0x0002 because retrieving a value stored in this manner would require 2 memory fetches instead of 1. *Rationale:* This is a necessary condition to assure that the SSP executes efficiently.
  - 16-bit values such as shorts and chars should not span 16-bit boundaries within the heap or ROM address space. *Rationale:* This is a necessary condition to assure that the SSP executes efficiently.

This list of hardware constraints is not meant to be exhaustive, but rather it demonstrates the nature of the constraints imposed by the SSP hardware. Taking a more rigorous approach, hardware constraints can be expressed in terms of a formula $\mathcal{H}$ in first-order logic. Abstractly speaking, this formula defines properties and relationships between (1) encoded symbolic references and addresses, and (2) addresses within an address space.

Given the formula $\mathcal{CH} \stackrel{def}{=} \mathcal{C} \wedge \mathcal{H}$ an interpretation $\mathcal{I}$ is a mapping from encoded symbolic references in $\mathcal{CH}$ to the domain $\mathcal{D} \stackrel{def}{=} D_{HEAP} \cup D_{ROM} \cup D_{CP} \cup D_{MT} \cup D_{OBJ}$ of absolute addresses and offset addresses. In this setting, resolution for the SSP can be defined as a function that constructs an interpretation $\mathcal{I}$ over $\mathcal{D}$ satisfying the formula $\mathcal{CH}$. From an operational perspective, the interpretation $\mathcal{I}$ is an assignment of address values to indexes (i.e., encoded symbolic references). In this article, the terms *resolution* and *resolve* are used to describe the processes behind the construction of such assignments.

We are now in a position to give a high-level definition of the class loader core that is the focus of this article.

Definition 1 *The* **core** *of the class loader for the SSP is an interpretation $\mathcal{I}_{core}$ mapping indexes to (1) offset addresses in $D_{MT} \cup D_{OBJ}$, and (2) absolute addresses in the space D_{HEAP} such that $\mathcal{I}_{core}$ satisfies $\mathcal{CH}$.*

Notice that the definition of the core, as we have defined it, excludes the offset address space D_{CP} and the absolute address space D_{ROM}. The reason for excluding these address spaces from consideration by the core is mainly for simplicity and space considerations. For example, calculating offsets for the constant pool entries is essentially the same as calculating offsets for method table entries. Since method table construction is included in the core the notion of calculating offsets for tables is already covered in the core. The calculation of absolute addresses for immutable objects in the ROM is also abstractly similar to table offset calculation and is also therefore omitted.

2 Overview of the Structure of Class Files

Figure 2 gives a top-level view of the components contained in a Java class file. Components such as cp_info, interfaces, field_info, and method_info are highly complex and contain of a number of subcomponents which are not shown in Figure 2. For more information about the structure of these components as well as class files in general see [10][19].

In this paper we restrict our attention to class loading activities for the SSP pertaining to the subset of Java class files shown in Figure 3. In particular, we will look at (1) index resolution, (2) static field address calculation, (3) offset address calculation, (4) method table construction, and (5) inter-class absolute address and offset address distribution. Collectively, we will refer to this set of class loading activities as the *class loader core*. In the following sections we informally describe each of these activities in detail.

2.1 Index Resolution

In Java class files, references to *field*, *method*, *this-class*, and *super-class* information are abstractly encoded as indexes into the class file's constant pool. Within a constant pool, such indexes directly or indirectly denote information that is ultimately expressed symbolically in terms of Utf8 strings. We will refer to the symbolic information denoted by an index as the *abstract meaning* otherwise known as the *symbolic reference* of that index. We use the term *index resolution* to denote the process of constructing the abstract meaning of indexes.

Consider the constant pool fragment shown in Figure 4. In this example, much detail has been abstracted away from the structure of the constant pool, and only the salient portions remain. For example, Java constant pool entries have tags that indicate the type of the entry, and Utf8 entries are realized in terms of a list of bytes. However, regardless of the presence or absence of such detail the concept of index resolution remains the same.

Given the abstracted constant pool in the Figure 4, we ask "What is the symbolic reference of the index 1 with respect to this constant pool"? To determine the meaning of an index, the chain of index/value pairs are followed until a collection of Utf8 values are reached. The concatenation of these Utf8 values forms the symbolic reference. The value at index 1 in the constant pool contains an entry of type *CONSTANT_Fieldref_info*, which indicates that the index 1 denotes an encoding of a symbolic reference to a field. When resolved this symbolic reference will consist of the name of the class in which the field is declared, the name of the field, and a descriptor describing the type of the field (e.g., integer, short, long). The contents of the *CONSTANT_Fieldref_info* entry at position 1 consists of two indexes.

magic	The hex value 0xCAFEBABE indicating that this file is a Java class file.
minor_version	The minor version of the compiler that produced this class.
major_version	The major version of the compiler that produced this class.
constant_pool_count	The number of entries in the constant pool.
cp_info	The constant pool.
access_flags	Modifiers associated with this class or interface (e.g., private, final, abstract, etc.).
this_class	A constant pool index that when resolved yields the name of the class.
super_class	The value 0 or a constant pool index that when resolved yields the name of the super class.
interfaces_count	The number of direct super interfaces of the class or interface.
interfaces	The interfaces implemented by the class.
fields_count	The number of fields explicitly declared in the class.
field_info	The fields explicitly declared in the class.
methods_count	The number of methods explicitly declared in the class.
method_info	The methods explicitly declared in the class.
attributes_count	The number of attributes of the class.
attribute_info	The attributes of the class.

Figure 2: The components of a Java class file

Java Structure Name	SSP Structure Name	Description
cp_info	*cp*	the constant pool
this_class	*this*	the name of this class
super_class	*super*	the name of the parent class
field_info	*sfields* & *ifields*	fields are separated into a list of static fields *sfields* and a list of instance fields *ifields*
method_info	*mt* & *methods*	*methods* are associated with a method table *mt*

Figure 3: The subset of the Java class file structure relevant to the class loader core

Index	Entry Type	Contents
1	CONSTANT_Fieldref_info	2 3
2	CONSTANT_Class_info	4
3	CONSTANT_NameAndType_info	5 6
4	CONSTANT_Utf8_info	B
5	CONSTANT_Utf8_info	y
6	CONSTANT_UTF8_info	I

Figure 4: A constant pool description of the integer field B.x

The Java class file specification requires that the first index be interpreted as a *class_index* denoting the class in which the field is declared. The second index should be interpreted as a *name_and_type_index* which provides information about the name and type of the field. Thus the first step in the index resolution process is: $1 \Rightarrow 2\ 3$. Next, index resolution is individually performed on the indexes 2 and 3. The constant pool entry having index 2 contains a *name_index* 4. Thus, the current state of the resolution is: $1 \Rightarrow 2\ 3\ \Rightarrow 4\ 3$. Now, the constant pool entry having index 4 contains an entry of type *CONSTANT_Utf8_info* whose value is "B". Thus, "B" is the name of the class in which the field denoted by index 1 is declared.

The second part of the value located at index 1 in the constant pool is the index 3. The constant pool entry at index 3 contains a *name_index* 5 that in this context denotes the name of the field, and a *descriptor_index* 6 denoting the type of the field. Thus, $3 \Rightarrow 5\ 6$. The contents of the constant pool at index 5 tells us that the name of the field is the $Utf8$ value "y". The contents at index 6 indicates that the field is of type "I". Thus, the symbolic reference of index 1 with respect to the given constant pool is $B\ y\ I$, that is, index 1 denotes the field y of type *integer* that is declared in class B. The index resolution of 1 can be summarized by the following rewrite sequence:

$$1 \Rightarrow 2\ 3\ \Rightarrow 4\ 3\ \Rightarrow B\ 3\ \Rightarrow B\ 5\ 6\ \Rightarrow B\ y\ 6\ \Rightarrow B\ y\ I$$

In general, index resolution concerns itself with the replacement of indexes with their symbolic references. The scope of this type of rewriting is limited to individual class files and is based solely on information found in the constant pool for the class. Complete details on the structure of constant pools can be found in the literature [10][19].

2.2 Static Field Address Calculation

The goal of static field address calculation is to assign a unique absolute address to each static field within a Java application. Since static fields are associated with a class rather than an object (i.e., an instance of a class), their number remains constant during runtime. Consider the Java application program shown in Figure 5 consisting of the classes B, C, and D .

If we assume a byte-addressable memory in the range 0x0000...0xFFFF, then the static fields in the application could be assigned to the absolute addresses as shown in Figure 6. From a semantic perspective this class loader activity can be seen as providing an interpretation (i.e., a concrete meaning) for the symbolic references of static fields.

```
class B   { ... static int x, b2; ... }
class C   { ... static int c1, x, c3; ... }
class D   { ... static int d1; ... }
```

Figure 5: Classes and their static fields

Static Field	Absolute Address
B.x	0x0000
B.b2	0x0004
C.c1	0x0008
C.x	0x000C
C.c3	0x0010
D.d1	0x0014

Figure 6: Mapping static fields to absolute heap addresses

2.3 Instance Field Offset Calculation

Instance fields, in contrast to static fields, are associated with objects rather than classes. Each object contains its own copy of every instance field declared in its corresponding class plus all of the instance fields inherited from its super class. Figure 7 shows the instance field declarations in a number of Java class fragments.

Figures 8 and 9 show possible offset calculations respectively for the instance fields of class C and E. We would like to point out that the class loader does not actually construct objects of the kind shown in Figures 8 and 9. That functionality is entrusted to the microcode implementation of the bytecode *new*. Instead, the purpose of the class loader is to construct an interpretation (i.e., assign a concrete semantics) in the form of a mapping from symbolic references of instance fields to object offset addresses in a manner that is consistent with the formula $\mathcal{CH}$ mentioned in Section 1.2. The interpretation provided by the object offset addresses must be sound with respect to all dynamic uses of objects. For example, in Java objects can be dynamically upcast and downcast. Thus all objects within an inheritance hierarchy must have a consistent interpretation for all shared (i.e., inherited) instance fields.

2.4 Method Table Construction

Encoded symbolic references to methods must ultimately be resolvable to the address where the bytecode for the method resides. However, this resolution is complicated by the interplay of two aspects of Java's subtype system. First, within an inheritance hierarchy multiple definitions for a single method may occur. Second, Java's *upcast* operation provides a mechanism by which the type of an object may be cast to that of any ancestor belonging to the inheritance chain.

Within such a fluid inheritance hierarchy, methods must utilize symbolic method references in a consistent fashion. Consistency here means that a symbolic reference to a method such as *B.foo* must

class B	{ ... int x, b2; ... }
class C extends B	{ ... int c1, x, c3; ... }
class D extends B	{ ... int d1; ... }
class E extends D	{ ... int x, e2; ... }

Figure 7: Classes and their instance fields

Object is Instance of C

Instance Field	**Offset from Base of Object**	**Comment**
B.x	0x0000	inherited from B
B.b2	0x0004	inherited from B
C.c1	0x0008	declared in C
C.x	0x000C	declared in C
C.c3	0x0010	declared in C

Figure 8: Mapping instance fields to object offsets

Object is Instance of E

Instance Field	**Offset from Base of Object**	**Comment**
B.x	0x0000	inherited from B
B.b2	0x0004	inherited from B
D.d1	0x0008	inherited from D
E.x	0x000C	declared in E
E.e2	0x0010	declared in E

Figure 9: Mapping instance fields to object offsets

have the ability to denote any redefinition of *foo* in every descendent of B. This capability is needed because during runtime, an object that is an instance of a descendent of B may be upcast to B after which the method *foo* could be invoked on the upcast object. In such a case, the constant pool of the class in which *B.foo* is invoked will contain a symbolic reference to *B.foo*. However, because multiple definitions of *foo* may exist throughout the inheritance hierarchy this symbolic reference cannot be directly resolved to a single absolute address. A standard solution to this problem is to construct a method table for each class [10][19]. This method table forms a layer of indirection that enables methods to be referenced in a consistent fashion. The entries in a method table contain data necessary to execute the bytecode corresponding to the implementation of a method as seen from the perspective of a particular class. For example, data in a method table may include the address of the first bytecode in the method as well as the address the method's corresponding constant pool. Symbolic references to methods such as *B.foo* are now resolved to offsets into the method table. Of course in order for the indirection provided by the method table solve our problem, all classes that inherit or redefine *foo* must

store data related to *foo* in the same relative position (i.e., offset) in their method table.

Figure 10 shows a class B and a class C which extends B. Note that although the method *foo* is redeclared in C, the method tables for B and C, shown in Figure 11, place the data for *foo* in the same relative location in their method tables. As a result, the symbolic reference to *B.foo* can be uniformly resolved to the method table offset 0x0008. Which method table this offset is applied to depends on the class from which an object is derived (e.g., B or C in Figure 10). For example, the expression *((B)(new C()))).foo()* will access the data at offset 0x0008 in C's method table while the expression *(new B()).foo()* will access the data at offset 0x0008 in B's method table.

```
class B              { ...
                       int f1(int x) {...}
                       int foo() {...}
                       int g1() {...}
                     ...}

class C extends B    { ...
                       int foo() {...}
                       int g2() {...}
                     ...}
```

Figure 10: Classes and their virtual methods

Offset	**Method Table for B**	**Comment**
0x0000	data for B.f1(I)I	declared
0x0008	data for B.foo()I	**declared**
0x0010	data for B.g1()I	declared

Offset	**Method Table for C**	**Comments**
0x0000	data for B.f1(I)I	inherited
0x0008	data for C.foo()I	**redeclared**
0x0010	data for B.g1()I	inherited
0x0018	data for C.g2()I	declared

Figure 11: Method table construction

2.5 Inter-class Absolute Address and Offset Address Distribution

Inter-class distribution is concerned with the distribution of absolute addresses and offset addresses between the various class files that make up a Java application. Within a single class file, symbolic references to locally declared fields and methods can be resolved to absolute addresses (for static fields), object offsets (for instance fields), and method table offsets (for methods). However, within a Java application, a class file X may have a symbolic reference to fields and methods that have been declared in another class file Y. References external to X show up as symbolic references in the constant pool of X and must be resolved using information originating from the class Y. Specifically, absolute address and offset address information must be distributed from the class in which the declarations occur (e.g., Y) to all classes referencing these declarations (e.g., X).

Figure 12 shows two class files B and C. The constant pool of class file B contains symbolic (non-local) references to a field and method declared in (local to) C, and the constant pool of C contains symbolic references to a field and method declared in B.

Class B				
Index	Constant Pool Entries	Fields	Method Table Offset	Method Table Info
1	*B.x I local*	*B.w I* 0x0000	0x00	*B.g (I)I ...*
2	*C.x I* ***non-local***	*B.x I* 0x0004	0x01	*B.f (I)I ...*
3	*B.f (I) local*	*B.y I* 0x0008	0x02	*B.foo (I)I ...*
4	*C.g (I)* ***non-local***	–	–	–
Class C				
Index	Constant Pool Entries	Fields	Method Table Offset	Method Table Info
1	*B.x I* ***non-local***	*C.x I* 0x0000	0x00	*C.bar (I)I ...*
2	*C.x I local*	*C.y I* 0x0004	0x01	*C.f (I)I ...*
3	*B.f (I)* ***non-local***	*C.z I* 0x0008	0x02	*C.g (I)I ...*
4	*C.g (I) local*	*C.k I* 0x000*C*	0x03	*C.h (I)I ...*

Figure 12: An examples of two classes having external symbolic references to fields and methods

Figure 13 shows the class files B and C after the *inter-class distribution* phase. Note that field address and method table offset data has been propagated between the classes.

Class B				
Index	Constant Pool Entries	Fields	Method Table Offset	Method Table Info
1	*B.x I* 0x0004	*B.w I* 0x0000	0x00	*B.g (I)I ...*
2	*C.x I* 0x0000	*B.x I* 0x0004	0x01	*B.f (I)I ...*
3	*B.f (I)* 0x01	*B.y I* 0x0008	0x02	*B.foo (I)I ...*
4	*C.g (I)* 0x02	–	–	–
Class C				
Index	Constant Pool Entries	Fields	Method Table Offset	Method Table Info
1	*B.x I* 0x0004	*C.x I* 0x0000	0x00	*C.bar (I)I ...*
2	*C.x I* 0x0004	*C.y I* 0x0004	0x01	*C.f (I)I ...*
3	*B.f (I)* 0x01	*C.z I* 0x0008	0x02	*C.g (I)I ...*
4	*C.g (I)* 0x02	*C.k I* 0x000*C*	0x03	*C.h (I)I ...*

Figure 13: Inter-class distribution of field offset/address and method table offsets

3 An overview of TL

The specification of the JVM enables class loading to occur dynamically (e.g., during runtime). However, from the perspective of class loading, the SSP can be considered a *closed system* because all the class files in an application must be stored on the ROM prior to execution. The closed nature of the SSP's

execution environment enables the class loading activities of the SSP to be performed statically, prior to execution. Under these conditions, the functionality of a class loader is well suited to a transformation-oriented implementation [25]. This section gives an overview of the strategic programming language TL which we will use to implement an abstract version of the class loader core.

In general, strategic programming systems are rewriting systems that have been extended with a variety of constructs enabling explicit control over the application of rewrite rules. Such control is typically necessary when dealing with rule sets that are non-confluent and/or non-terminating. When constructed appropriately, rewriting systems and strategic programming systems have properties that make them well-suited to formal verification. Our research goal into the development of TL has been to provide a framework whereby strategic programs can be written to realize complex functionality in a manner which nevertheless remains within the grasp of formal automated verification.

3.1 The Basic Constructs of TL

TL is a higher-order strategic programming language [26][29][27][28]. In TL, conditional rewrite rules can be combined to form expressions called *strategies*. Strategies define controlled sequences of rewrites and can be applied to tree structures to produce other tree structures. Thus, a strategy can be viewed as a function that rewrites or *transforms* one tree into another.

The primary constructs and abstractions in a first-order strategic programming language typically include:

1. *patterns* – A *pattern* is a notation for describing the tree structures that are being manipulated. This notation typically includes variables, potentially typed, that are quantified over a variety of tree structures.

2. *rewrite rules* – A *rewrite rule* is a construct for specifying that one *pattern* is to be replaced by another *pattern*. In a strategic system, *rewrite rules* are also considered to be a degenerative form of *strategy*.

3. *conditions* – A *condition* is a construct associated with a rewrite rule that restricts its application.

4. *combinators* – A *combinator* is an operator (generally unary or binary) that can be used to compose one or more *strategies* into a new *strategy*.

5. *generic traversals* – A *generic traversal* can be thought of as a curried function parameterized on a strategy s and a tree t. As the name suggests, a generic traversal will traverse its input tree structure t and apply its input strategy s at one or more points along the traversal. A typical and very useful generic traversal is one that performs a top-down left-to-right traversal of a tree structure and uniformly applies its input strategy to all sub-trees encountered.

6. *strategies* – In its purest sense, a first-order *strategy* can be characterized as any function that transforms one tree into another tree. Structurally speaking however, a *strategy* is an expression composed of rewrite rules, combinators, and generic traversals.

7. *labels* – A strategy can be bound to a *label* for the purposes of abstraction.

In addition to the first-order constructs mentioned above, TL also supports the following higher-order constructs:

1. *higher-order strategies* – A *higher-order strategy* is a strategy that when applied to a tree will return a strategy rather than a tree. For example, when applied to a tree, a second-order strategy s^2 will yield a first-order strategy s^1. In general, the application of a strategy of order n to a term t will yield a strategy of order $n-1$. For the purposes of uniformity, in this framework, a tree is considered to be a strategy of order 0.

2. *higher-order generic traversals* – A *higher-order generic traversal* can be thought of as a curried function that is parameterized on a higher-order strategy s^n (where n denotes the order of the strategy) and a tree t. Its application to t yields a strategy of order $n-1$. Higher-order generic traversals are very useful for creating specific strategies that are "tuned" the data occurring within a particular tree. In other words, they are a mechanism for dynamically generating strategies customized for a given input tree.

In the following sections we briefly describe each of the constructs mentioned above.

3.2 Tree/Term Notation

Let $G = (N, T, P, S)$ denote a context-free grammar where N is the set of nonterminals, T is the set of terminals, P is the set of productions, and S is the start symbol. Given an arbitrary symbol $B \in N$ and a string of symbols $\alpha = X_1X_2...X_m$ where for all $1 \leq i \leq m : X_i \in N \cup T$, we say B derives α iff the productions in P can be used to expand B to α. Traditionally, the expression $B \overset{*}{\Rightarrow} \alpha$ is used to denote that B can derive α in zero or more expansion steps. Similarly, one can write $B \overset{+}{\Rightarrow} \alpha$ to denote a derivation consisting of one or more expansion steps.

In TL, we write $B[[\alpha']]$ to denote an *instance* of the derivation $B \overset{+}{\Rightarrow} \alpha$ whose resulting value is a parse tree having B as its root symbol. In TL, expressions of the form $B[[\alpha']]$ are referred to as *parse expressions*. In the parse expression $B[[\alpha']]$ the string α' is an *instance* of α because nonterminal symbols in α' are constrained through the use of subscripts. Subscripted nonterminal symbols are referred to as *schema variables* or simply *variables* for short. TL also considers a schema variable (e.g., B_i) to be a parse expression in its own right. Within a rewrite rule all occurrences of schema variables having the same subscript denote the same variable.

Figure 14 shows a BNF grammar fragment describing a small portion of an imperative language. In the context of this grammar, the parse expressions $stmt[[\ id_1 = 5\ ]]$ and $stmt[[\ id_2 = 5\ ]]$ both describe instances of the derivation $stmt \overset{+}{\Rightarrow} id\ =\ 5$.

When the specific structure of a parse expression is unimportant the parse expression will be denoted by variables of the form $t, t_1, ...$ or variables of the form $tree, tree_1, tree_2$, and so on. Parse expressions containing no schema variables are called *ground* and parse expressions containing one or more schema variables are called *non-ground*. And finally, within the context of rewriting or strategic programming, *trees* as described here can and generally are viewed as *terms*. When the distinction is unimportant, we will refer to *trees* and *terms* interchangeably.

prog	::=	*stmt_list*
stmt_list	::=	*stmt* "; " *stmt_list* | *stmt*
stmt	::=	*assign* | *cond* | ...
assign	::=	*lvalue* "=" *expr*
...		
lvalue	::=	*id*
expr	::=	*int* | *int* + *int*
...		

Figure 14: A concrete syntax fragment

3.3 Conditional Rewrite Rules

Figure 15 defines the structure of TL rules in terms of an extended-BNF. The meta-symbols of this BNF are [,] and ::=. Symbols enclosed in square brackets denote optional portions of a production. For example, rewrite rules in TL have the form $lhs \rightarrow rhs$ and may have optional labels and conditions associated with them.

rule	::=	[*label* :] $lhs \rightarrow rhs$ [if *condition*]
label	::=	identifier
lhs	::=	a parse expression
rhs	::=	a strategic expression whose evaluation yields a parse tree

Figure 15: The syntactic structure of rewrite rules in TL

Based on the tree grammar given in Figure 14, we can write the following condition-less rewrite rule:

$$r_1 : stmt[[id_1 = 4 + 1]] \rightarrow stmt[[id_1 = 5]]$$

This rule states that an assignment statement having the expression $4 + 1$ as its right-hand side should be rewritten to an assignment statement having the constant 5 as its right-hand side.

3.3.1 Conditions

The conditional portion of a rule is a *match expression* consisting of one or more *match equations*. The symbol $\ll$, adapted from the ρ-calculus [6], is used to denote first-order matching modulo an empty equational theory. Let t_2 denote a ground tree and let t_1 denote a parse expression which may contain one or more schema variables. A *match equation* is denoted $t_1 \ll t_2$. 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 a boolean valued test for syntactic equality.

A *match expression* is a boolean expression involving one or more match equations. 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.

Based on the tree grammar given in Figure 14, we can write the following conditional rewrite rule:

$$r_2 : stmt_1 \rightarrow stmt[[id_1 = 5]] \ \textit{if} \ stmt_1 \ll stmt[[id_1 = 4 + 1]]$$

The conditional rule r_2 says that a statement $stmt_1$ should be rewritten to the statement $stmt[[id_1 = 5]]$ only if the condition $stmt_1 \ll stmt[[id_1 = 4 + 1]]$ is satisfied. In other words, if $stmt_1$ is an assignment statement whose right-hand side is the expression $4 + 1$.

The rule r_3 shown below gives a trivial example of a rule condition consisting of two match equations.

$$r_3 : stmt_1 \rightarrow stmt[[id_1 = 5]] \ \textit{if} \ stmt_1 \ll stmt[[id_1 = expr_1]] \wedge expr_1 \ll expr[[4 + 1]]$$

The conditional rule r_3 says that a statement $stmt_1$ should be rewritten to the statement $stmt[[id_1 = 5]]$ only if the condition $stmt_1$ is an assignment statement whose right-hand side is $expr_1$ and $expr_1$ is $4+1$.

3.3.2 Rule Application

The application of a conditional rewrite rule r to a tree t is expressed as $r(t)$ where r is either an abstraction of a rewrite rule (i.e., a label) or an anonymous rule value e.g., $lhs \rightarrow rhs$. We adopt a curried notation in the style of ML where application is a left-associative implicit operator and parentheses are 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 $r(t)$.

Let us consider the rules r_1, r_2, and r_3 described in the previous sections. The application $r_1\ stmt[[x = 4{+}1]]$ yields $stmt[[x = 5]]$ as do the applications $r_2\ stmt[[x = 4{+}1]]$ and $r_3\ stmt[[x = 4{+}1]]$. In contrast, the application $r_1\ stmt[[y = 6]]$ is said to *fail* and in TL will yield $stmt[[y = 6]]$ as its result.

Let $r : lhs \rightarrow rhs \ \textit{if condition}$ denote an abstract rule. In TL, the application of r to t proceeds as follows: First, the match-equation $lhs \ll t$ is evaluated. If $lhs \ll t$ evaluates to false, the rule application *fails* and t is left unchanged. Otherwise, $lhs \ll t$ evaluates to true and the computation proceeds to the evaluation of the *condition* associated with r. Again, if this *condition* evaluates to false, the rule application *fails* and t is returned unchanged. Otherwise the rule application *succeeds* and the value of rhs is returned.

3.4 Combinators

TL provides three binary combinators enabling (1) the sequential composition of strategies which is denoted by the semi-colon symbol, (2) the left-biased composition of strategies which is denoted by the symbol $<+$, and (3) the right-biased composition of strategies which is denoted by the symbol $+>$.

Let s_1 and s_2 denote two first-order strategies. The strategy $s_1; s_2$ denotes the sequential composition of s_1 and s_2. When applied to a term t the strategy $s_1; s_2$ will first apply s_1 to t yielding t' and then apply s_2 to t' yielding t''.

The strategy $s_1 <+ s_2$ denotes the left-biased composition of s_1 and s_2. When applied to a term t the strategy $s_1 <+ s_2$ will first try to apply s_1 to t. If the application $s_1\ t$ succeeds, and produces t' as

its result, then t' is returned as the final result and s_2 is not used. However, if the application s_1 t fails, then the result of s_2 t is returned as the final result.

The strategy s_1 +> s_2 denotes the right-biased composition of s_1 and s_2. Semantically speaking, s_1 +> s_2 is equivalent to s_2 <+ s_1.

In addition to the three binary combinators previously mentioned, TL also provides the unary combinators *transient* and *hide*. The semantics of these combinators are described in Section 4.2.

3.5 Generic First-Order Traversals

TL supports a number of generic first-order traversals including: TDL, FIX_TDL, and TDL_B. The traversal TDL accepts a first-order strategy s and a tree t as input and performs a single top-down left-to-right traversal over t applying s to every sub-tree encountered. The traversal FIX_TDL accepts a first-order strategy s and a tree t_0 and performs the evaluation TDL s t_0 yielding t_1. If one or more rewrites occurred during the evaluation of TDL s t_0, then FIX_TDL will perform the evaluation TDL s t_1 yielding t_2. Additional evaluations of the form TDL s t_i will continue until a tree t_j is reached such that the evaluation TDL s t_j completes without a single rewrite being performed on t_j. If such a tree t_j is found, then this is the value of the evaluation of FIX_TDL s t_0. Otherwise the evaluation of FIX_TDL s t_0 does not terminate.

The generic first-order traversal TDL_B is described in Section 4.4.

3.6 Higher-Order Rules and Strategies

In TL, a conditional rewrite rule of order $n+1$ has the form:

$$label : lhs \rightarrow s^n \ \textit{if condition}$$

where s^n is a strategic expression whose evaluation yields a strategy of order n. As was the case with first-order rules, the label and conditional portion of higher-order rules are optional.

The combinators in TL can be applied to first-order as well as higher-order strategies. For example, let s_1^n and s_2^n denote two order n strategies. The expressions $s_1^n; s_2^n$ and s_1^n <+ s_2^n respectively denote the sequential and left-biased conditional composition of s_1^n and s_2^n. For various practical reasons, TL imposes the restriction that only strategies having the same order may be composed with one another.

3.7 Higher-Order Generic Traversals

TL supports a number of generic higher-order traversals including: *seq_tdl* and *lcond_tdl*. The traversal *seq_tdl* accepts a higher-order strategy s^n and a tree t_1 as its input and performs a top-down left-to-right traversal of t_1 applying the strategy s^n to each (sub)tree encountered. Let t_1, t_2, ..., t_m denote the trees encountered during the traversal of t_1. Let s_i^{n-1} denote the strategy obtained from applying s^n to the tree t_i. Given these assumptions, the evaluation of *seq_tdl* s^n t_1 will produce the strategy

$$s_1^{n-1}; s_2^{n-1}; ...; s_m^{n-1}$$

Similarly, the evaluation of *lcond_tdl* s^n t_1 will produce the strategy

$$s_1^{n-1} <+ s_2^{n-1} <+ ... <+ s_m^{n-1}$$

Having described the basics of TL we are now in a position to discuss a strategic implementation of the class loader core.

4 A Strategic Implementation of the Class Loader Core

In this section we look at a strategic solution, written in TL, to an abstract version of the class loader *core* as defined in Section 1.2. In this abstract example, the structure of a class file has been greatly simplified and in several places its structure has even been altered. In spite of these changes, we nevertheless make the claim that, when seen from a strategic perspective, the abstractions below still contain the essence of the class loader core. This claim is based on our experience in successfully developing a strategic-based implementation of an actual class loader for the SSP. This implementation has been developed using the HATS system, an IDE for strategic programming supporting a restricted version of TL in dialect form. The HATS system is freely available [7].

Noteworthy characteristics of the class file structure described in Figure 16 include:

- Additional terminal symbols have been added to enable the class file structure to be described by a context-free grammar.
  - Constant pool entries have been enclosed in parenthesis.
  - The @ symbol is used to tag absolute addresses belonging to D_{HEAP}.
  - The : symbol is used to tag object offset addresses belonging to D_{Obj}.
  - The # symbol is used to tag method table offset addresses belonging to D_{MT}.
- Explicit indexes have been given to the entries in the constant pool.
- The structure of a class file has been extended with a children list which is initially empty.
- Fields have been partitioned into a list of static fields and a list of instance fields.
- The methods section consists of a method table and a list of methods.
- Methods consist only of a method name. In particular, they do not contain a descriptor and they do not contain any bytecodes.
- All fields are tacitly assumed to be of type integer.
- Fields are denoted by a single index whose resolution yields the name of the field and the class in which it is declared.
- Address and offset units are words.

In the context of these constraints and alterations we present a strategic implementation of the class loader core. Figure 17 shows three abstract class files before class loading, and Figure 18 shows the same three class files after class loading. In order to construct a sufficiently rich example that is nevertheless small in size, the class file examples shown are not proper class files. For example, a super type makes a reference to its subtype.

app	::=	app class | ϵ
class	::=	{ class_id parent_id info children }
children	::=	children class | ϵ
info	::=	cp fields methods
class_id	::=	id
parent_id	::=	id
cp	::=	cp c_entry | ϵ
c_entry	::=	(index , data)
fields	::=	statics instance
statics	::=	statics sfield | ϵ
sfield	::=	data @ addr
instance	::=	instance ifield | ϵ
ifield	::=	data : addr
methods	::=	mt , method_list
mt	::=	mt_entry mt | ϵ
mt_entry	::=	key # addr
method_list	::=	m_entry method_list | ϵ
m_entry	::=	data ( )
data	::=	key | d | key address_type addr
key	::=	d . d
d	::=	id | index
address_type	::=	@ | # | :
index	::=	integer
addr	::=	integer
id	::=	ident

Figure 16: An extended-BNF grammar describing a simplified application in terms of a list of class files

This class	1
Super class	19
CP	(1,A)(2, 1.3)(3, x1)(4, 1.5)(5, x2)(6, 1.8)(7, 1.9)(8,a1)(9, a2)(10, 11.3) (11, B)(12, 11.13)(13, foo)(14, x3)(15, bar)(16, 1.14)(17,1.13)(18,1.15)(19, Obj)
Static fields	2@- 4@- 16@-
Instance fields	6:- 7:-
MT	
Methods	17() 18()
This class	3
Super class	19
CP	(1, x1)(2, 3.1)(3, B)(4, x2)(5, 3.4)(6, x3)(7,3.6)(8, b1)(9, 3.8)(10, b2) (11, 3.10)(12, foo)(13, 3.12)(14, f)(15, 3.14)(16, C)(17, 16.1)(18, 16.4)(19,A)
Static fields	2@- 5@- 7@-
Instance fields	9:- 11:-
MT	
Methods	13() 15()
This class	3
Super class	21
CP	(1, x1)(2, 3.1)(3, C)(4, x2)(5, 3.4)(6, x3) (7, 3.6)(8, c1)(9, 3.8)(10, c2) (11, 3.10)(12, bar)(13, 3.12)(14, f)(15, 3.14)(16, B)(17, 16.1)(18,16.4) (19, b1)(19, 16.19)(20, b2)(21, 16.20)(21,A)
Static fields	2@- 5@- 7@-
Instance fields	9:- 11:-
MT	
Methods	13() 15()

Figure 17: Three abstract class files prior to class loading

This class	A
Super class	Obj
CP	(1, A)(2, A.x1@0)(3, x1)(4, A.x2@1)(5, x2)(6, A.a1:0)(7, A.a2:1)(8, a1) (9, a2)(10, B.x1@3)(11, B)(12, B.foo#0)(13, foo)(14, x3)(15, bar) (16, A.x3@2)(17, A.foo#0)(18, A.bar#1)(19, Obj)
Static fields	A.x1@0 A.x2@1 A.x3@2
Instance fields	A.a1:0 A.a2:1
MT	A.foo#0 A.bar#1
Methods	A.foo() A.bar()
This class	B
Super class	A
CP	(1, x1)(2, B.x1@3)(3, B)(4, x2)(5, B.x2@4)(6, x3)(7, B.x2@5)(8, b1) (9, B.b1:2)(10, b2)(11, B.b2:3)(12, foo)(13, B.foo#0)(14, f) (15, B.f#2)(16, C)(17, C.x1@6)(18, C.x2@7)(19,A)
Static fields	B.x1@3 B.x2@4 B.x3@5
Instance fields	B.b1:2 B.b2:3
MT	B.foo#0 A.bar#1 B.f#2
Methods	B.foo() B.f()
This class	C
Super class	A
CP	(1, x1)(2, C.x1@6)(3, C)(4, x2)(5, C.x2@7)(6, x3) (7, C.x3@8)(8, c1) (9, C.c1:2)(10, c2)(11, C:x2:3)(12, bar)(13, C.bar#1)(14, f) (15, C.f#2)(16, B)(17, B.x1@3)(18,B.x2@4) (19, b1)(19, B.b1:2)(20, b2)(21, B.b2:3)(21,A)
Static fields	C.x1@6 C.x2@7 C.x3@8
Instance fields	C.c1:2 C.c2:3
MT	A.foo#0 C.bar#1 C.f#2
Methods	C.bar() C.f()

Figure 18: The three abstract class files shown in Figure 17 after class loading

index_resolution	:	$class_0 \rightarrow$ *FIX_TDL* (*seq_tdl cp_normalize* $class_0$) $class_0$
cp_normalize	:	*c_entry*[[($index_1$, d_1)]] $\rightarrow$ d[[$index_1$]] $\rightarrow$ d_1

Figure 19: Index Resolution

4.1 Index Resolution in TL

As described in Section 2.1, the goal of index resolution is to rewrite constant pool indexes into their symbolic references. In the abstract example that we are considering a constant pool entry has the form ($index_1$, $data_1$), where $index_1$ denotes the position of the entry and $data_1$ denotes its value (e.g., an encoding of a symbolic reference or a symbolic reference). Conceptually speaking, our strategic approach to index resolution is as follows. For every constant pool entry ($index_1$, $data_1$) we would like to create a rewrite rule of the form $index_1 \rightarrow data_1$. Once this has been accomplished all that remains to be done is to exhaustively apply the resulting rules to all *data indexes* within the class file. We would like to point out that the structure of the constant pool entries ($index_1$, $data_1$) is such that $index_1$, the positional denotation of the entry, is not in-and-of-itself a *data index*. *Data indexes* occur in a variety of places throughout a class file such as within the description of fields, within the description of methods including their bytecodes, and within the data portion of constant pool entries.

The grammar in Figure 16 has been defined in such a way to enable an ordinary use of an *index* to be distinguished from a *data index*. In particular, all data indexes are derived from the nonterminal symbol d. Thus, it is only in this structural context that index resolution should be performed. This constraint is captured by the parse expression d[[$index_1$]], yielding the rewrite rule d[[$index_1$]] $\rightarrow$ d_1.

Figure 19 gives an implementation of index resolution in TL. The behavior of the strategy *index_resolution* is as follows. When applied to a class file structure $class_0$ the strategy *index_resolution* will first evaluate the strategic expression *seq_tdl cp_normalize* $class_0$. Within this expression, the strategy *seq_tdl* is a higher-order generic traversal that will traverse a term in a top-down left-to-right (tdl) fashion. In this case, the term being traversed is $class_0$. The fact that *seq_tdl* is higher-order means that it expects to apply a higher-order strategy to the sub-terms of the term it is traversing. In this case, the higher-order strategy being applied is *cp_normalize*, a second-order strategy that converts a constant pool entry of the form *c_entry*[[($index_1$, d_1)]] into a first-order rewrite rule of the form: d[[$index_1$]] $\rightarrow$ d_1. When applied to the entries of the constant pool of $class_0$ a number of instances of the rule d[[$index_1$]] $\rightarrow$ d_1 will be generated. These rule instances are then composed by *seq_tdl* using TL's **seq**uential composition operator. This composition is part of the semantics of *seq_tdl* – which sequentially composes the results generated generates from its tdl traversal. The resulting first-order strategy is of the form $r_1; r_2; ...r_n$ where r_i is the instance of d[[$index_1$]] $\rightarrow$ d_1 corresponding to the ith constant pool entry. This first-order strategy is then exhaustively applied to $class_0$ in a top-down left-to-right fashion by the first order generic traversal *FIX_TDL*. The result is that all *data indexes* are rewritten to their symbolic references.

4.2 Static Field Address Calculation in TL

As described in Section 2.2, the goal of static field address calculation is to assign each static field in an application a unique absolute address taken from the address space D_{HEAP}. In TL, this can be accomplished with a strategy that makes use of the strategic combinators *transient* [26][27]and *hide* [28][29]. Both of these combinators are unique to TL.

When applied to a strategy s, the *transient* combinator has the effect of "erasing" s after the first successful application of s. Thus, the *transient* combinator can be used to create a strategy that can be applied at most once. For example, let s_1 denote the strategy $transient(addr[[\ 0\ ]] \rightarrow addr[[\ 1\ ]])$. The strategic expression $TDL\ s_1\ class_0$ states that the term $class_0$ should be traversed in a top-down left-to-right fashion and that the **current value** of strategy s_1 should be applied to every term encountered during this traversal. Due to the use of the *transient* combinator in s_1, the strategic expression $TDL\ s_1\ class_0$ will have the effect of rewriting the first term matching $addr[[\ 0\ ]]$ to $addr[[\ 1\ ]]$ and will leave unchanged all other terms.

Generalizing this example, let s_i denote a strategy of the form $transient(addr[[\ 0\ ]] \rightarrow addr[[\ i\ ]])$, and let $s_{1..n}$ denote a strategy of the form $s_1; s_2; ...; s_n$. The strategic expression $TDL\ s_{1..n}\ class_0$ will rewrite the first occurrence of $addr[[\ 0\ ]]$ to $addr[[\ 1\ ]]$, the second occurrence of $addr[[\ 0\ ]]$ to $addr[[\ 2\ ]]$, and the nth occurrence of $addr[[\ 0\ ]]$ to $addr[[\ n\ ]]$. Suppose we are given a Java application whose class files collectively contain m static fields, all of which are of type integer and whose initial default address has been set to 0. When controlled properly, the application of a strategy of the form $s_{1..m}$ can be used to correctly assign a unique address to each static field in the application. In spirit, this is the transformational effect that we want to accomplish. However, for a variety of reasons, in practice it is difficult to attempt to construct such a strategy directly. Thus, we will use the *hide* combinator to construct a slightly different strategy whose net effect is equivalent to $s_{1..m}$.

In strategic programming, the left-biased choice combinator $<+$ is used to specify the conditional application of two or more strategies. When applied to a term t, the strategy $s_1 <+ s_2$ specifies that first the application of s_1 to t should be attempted. If this application succeeds and yields t', then the result of $(s_1 <+ s_2)\ t$ is t'. On the other hand, if the application of s_1 to t fails, then the left-biased choice combinator indicates that the application of s_2 to t should be tried next. In light of this, let us consider the strategy $hide(s_1) <+ s_2$. The *hide* combinator is a combinator that restricts the ability of the left-biased (or right-biased) choice combinator to observe whether or not the application of s_1 has succeeded or failed. More specifically, the *hide* combinator always gives the left-biased choice combinator the impression that the application of s_1 has failed. Thus, the strategy $hide(s_1) <+ s_2$ is equivalent to $s_1; s_2$.

When considered in isolation, the *hide* combinator is not very interesting. However, when combined with the *transient* combinator it becomes possible to construct strategies having interesting behaviors, such as a strategy that implements a sum. For example, consider the following strategy:

$$sum = \begin{cases} hide(addr[[\ i\ ]] \rightarrow addr[[i+1]]) <+\ transient(addr_1 \rightarrow addr_1) <+ \\ hide(addr[[\ i\ ]] \rightarrow addr[[i+1]]) <+\ transient(addr_2 \rightarrow addr_2) <+ \\ hide(addr[[\ i\ ]] \rightarrow addr[[i+1]]) <+\ transient(addr_3 \rightarrow addr_3) \end{cases}$$

When applied to a term t, the strategy *sum* will increment the first *addr* it encounters by one, the second *addr* it encounters by two (i.e., 1 + 1), and the third *addr* it encounters by three (i.e., 1 + 1 + 1). All remaining instances of *addr* encountered will be incremented by three. The informal explanation of this is as follows. Within t, when applied to the first occurrence of *addr*, the first *hide* encapsulated rule in *sum* will apply, incrementing the address by one. Due to the encapsulation of the *hide* combinator, the conditional operator <+ immediately to the right of the *hide* strategy is led to believe that the application of $hide(addr[[\ i\ ]] \rightarrow addr[[\ i+1\ ]])$ failed. Thus, the application of the next strategy within *sum* is attempted, which in this case is $transient(addr_1 \rightarrow addr_1)$. The rule $addr_1 \rightarrow addr_1$ is an identity that rewrites the term $addr_1$ to itself. From the perspective of <+ the application of the *transient* encapsulated rule succeeds, completing the application of *sum* to *addr*. However, since *transient* strategies can only be applied once, all future applications of this first *transient* strategy will fail, which in the context of *sum* will cause the application of *sum* to move on to its next rule. Thus, when *sum* is applied to a second *addr*, the first two *hide* encapsulated rules will apply, after which the second *transient* encapsulated rule is encountered (for the first time), which again completes the application of *sum* to the second *addr*. By similar reasoning, the application of *sum* to a third *addr* will result in three *hide* encapsulated increments.

static_addresses	:	$app_0 \rightarrow TDL(\ lcond_tdl\ sfield_sum\ app_0)\ app_0$
sfield_sum	:	$sfield_0 \rightarrow$
		$hide(sfield[[key_1\ @\ addr_1]] \rightarrow sfield[[key_1\ @\ addr_1+1]]\ )\ <+\ transient(sfield_1 \rightarrow sfield_1)$

Figure 20: A TL strategy for absolution address calculation

Figure 20 gives a TL implementation of static field address calculation. The absolute address for static fields are realized via a strategic sum that is based on the increment technique described by the *sum* strategy discussed previously. In particular, the higher order strategy *sfield_sum* performs a sum that applies only to static fields. Within the body of *static_addresses*, the higher-order strategic expression *lcond_tdl sfield_sum* app_0 will traverse app_0 in a tdl fashion and apply the strategy *sfield_sum*. This will produce one instance of the strategy

$$hide(sfield[[key_1\ @\ addr_1]] \rightarrow sfield[[key_1\ @\ addr_1+1]]\ )\ <+\ transient(sfield_1 \rightarrow sfield_1)$$

for each static field encountered. These strategy instances are then combined using the left-biased choice combinator. The resulting composition is an sfield sum which is then applied to app_0 using the generic traversal TDL. This has the effect of assigning a unique absolute offset to each static field in app_0.

4.3 Instance Field Offset Calculation in TL

Instance field offset calculation (see Section 2.3) is similar to static field address calculation. The primary difference is that the assignment of offset addresses is constrained by the subtype (i.e., inheritance) relationships between the class files within an application. In static field address calculation, the static fields of all classes in the application could be collected (in any order) and aggregated into a sum, which could then be applied to the entire application (in the same order as collected). In instance field offset

calculation, the instance fields of each class in an inheritance chain must be collected according to the order/position of the class in the inheritance chain. The resulting sum must then be applied only to the instance fields of that chain.

In TL, there are several ways to construct strategies whose application is restricted to individual inheritance chains. The approach taken in this article requires that an application first be restructured so that the new structure explicitly reflects the inheritance hierarchy of the class files. This is accomplished by adding a *children* element to each class file (see Figure 16). This children structure denotes a list of class files and creates the possibility of restructuring the class files from an application (which are initially list form) into an inheritance tree. After this has been accomplished an iterative process can be specified whereby a strategy realizing a partial sum is created using the instance fields belonging to a given class file $class_1$ and the resulting partial sum strategy is applied to all the classes belonging to the inheritance tree having $class_1$ as its root (i.e., $class_1$ and all its descendants). During the course of a top-down traversal, each class file will in turn become the root of its own inheritance (sub)tree and have a partial sum created for it. The cumulative result is that each instance field will eventually be assigned a proper offset address (i.e., a fully totaled sum).

instance_offsets	:	$app_1 \rightarrow app_3$
	if	$app_2 \ll TDL\ (seq_tdl\ create_hierarchy\ app_1)\ app[[\ \{Obj\ Obj\ \}\ ]]$
	$\wedge$	$app_3 \ll TDL\ sum_offsets\ app_2$
create_hierarchy	:	$class_1 \rightarrow class[[\ \{id_2\ id_3\ info_2\ children_2\}\ ]] \rightarrow class[[\ \{id_2\ id_3\ info_2\ children_2\ class_1\}\ ]]$
	if	$class_1 \ll class[[\ \{id_1\ id_2\ info_1\ children_1\}\ ]]$
sum_offsets	:	$class_0 \rightarrow TDL\ (lcond_tdl\ partial_sum\ instance_1\ )\ class_0$
	if	$class_0 \ll class[[\ \{class_id_1\ parent_id_1\ cp_1\ statics_1\ instance_1\ methods_1\ children_1\}\ ]]$
partial_sum	:	$class_0 \rightarrow TDL(\ lcond_tdl\ local_ifield_sum\ class_0)\ class_0$
local_ifield_sum	:	$ifield_0 \rightarrow$
		$hide(\ ifield[[key_1 : addr_1]] \rightarrow ifield[[key_1 : addr_1 + 1]]\ ) <+\ transient(ifield_1 \rightarrow ifield_1)$

Figure 21: Offset address calculation

Figure 21 gives a TL strategy that implements the approach to instance offset calculation just described. When applied to app_1, the strategy *instance_offsets* will first restructure app_1 into an inheritance tree. The evaluation of the strategic expression *seq_tdl create_hierarchy* app_1 will perform a top-down left-to-right traversal on app_1 and apply the higher-order strategy *create_hierarchy* to each class file encountered. The application of *create_hierarchy* to $class_1$ will produce a first-order strategy that places $class_1$ into the children list of its parent class. Notice that in the strategy *create_hierarchy*, id_1 and id_2 respectively denote the class name and parent class name of $class_1$. By definition, the parent class of $class_1$ is that class which has id_2 as its class name. Thus, in *create_hierarchy* the strategy

$$class[[\ \{id_2\ id_3\ info_2\ children_2\}\ ]] \rightarrow class[[\ \{id_2\ id_3\ info_2\ children_2\ class_1\}\ ]]$$

will place $class_1$ into the children list of its parent class.

For each class in app_1 such a first-order strategy is created and the resulting strategies are sequentially composed. The resulting composition is then applied in a top-down left-to-right fashion to an application structure consisting of the single class Obj, which is initially empty (i.e., Obj contains no constant pool, no fields, no methods, and no children). This application has the effect of *growing* an inheritance tree rooted at Obj. For example, first the children of Obj will be inserted into the children list of Obj. In turn, the children of Obj will have their children inserted into their children list, and so on. The resulting structure is then bound to app_2 via a match equation.

After the application has been restructured into an inheritance tree, the calculation of instance field offsets can begin. In the strategy *instance_offsets*, the evaluation of the strategic expression TDL *sum_offsets* app_2 will traverse the inheritance tree app_2 in a top-down left-to-right fashion and apply the strategy *sum_offsets* to each class file encountered. In turn, the strategy *sum_offsets* will traverse the instance fields of the class file $class_1$ to which it is applied and create an instance of *local_ifield_counter* for each field encountered. The instances of *local_ifield_counter* are then conditionally composed and the resulting strategy is applied to in a top-down left-to-right fashion to the inheritance tree rooted at $class_1$. This has the effect of (1) assigning proper (i.e., completed) offsets for the instance fields in $class_1$, and (2) assigning partially completed offsets for the instance fields of all the classes which are descendants of $class_1$. In general, the partially completed offsets for instance fields local to a given $class_i$ are completed when the strategy *sum_offsets* is applied to $class_i$ (i.e., it is treated as the root of an inheritance tree).

4.4 Method Table Construction in TL

Method table construction (see Section 2.4) is similar to instance field offset calculation in the sense that strategies are applied along inheritance chains. In the case of method table construction the goal is to construct a method table for each class within an application. The general algorithm implemented is as follows. One begins at the class Obj with an empty method table. This table is then propagated to all the descendants of Obj after which, method table construction begins for the children of class Obj. Let $class_1$ denote a arbitrary child of class Obj. The methods declared in $class_1$ are "added" to the method table of $class_1$ as well as to the method tables of all the descendants of $class_1$. This distribution represents the methods that are initially inherited by the descendants of $class_1$. In general, the methods in $class_i$ are added to the method table of $class_i$ and all of its descendants. The "addition" of a method m to the method table of $class_i$ can take one of the following three forms:

1. A method table entry m' is encountered corresponding to the method m. This means that m' has been redefined by m in $class_i$. In this case, m replaces (overwrites) the entry for m'.

2. The last entry of a (non-empty) method table is reached without encountering an entry corresponding to m. This means that, m is a new (i.e., previously unseen) method declared in $class_i$. In this case, a new entry is added to the end of the method table and its offset is assigned the value of the previous offset plus one.

3. An empty method table is encountered. In this case, an entry for m is added to the method table and given an offset of zero.

$$
\begin{array}{lcl}
\mathit{mt_construction} & : & \mathit{app}_1 \rightarrow \mathit{TDL}\ \mathit{add_methods}\ \mathit{app}_1 \\
\\
\mathit{add_methods} & : & \mathit{class}_1 \rightarrow TDL_B\ (\mathit{seq_tdl}\ \mathit{insert_method}\ \mathit{method_list}_1)\ \mathit{class}_1 \\
 & \text{if} & \mathit{class}_1 \ll \mathit{class}[[\ \{\mathit{class_id}_1\ \mathit{parent_id}_1\ \mathit{cp}_1\ \mathit{fields}_1\ \mathit{mt}_1\ \mathit{method_list}_1\ \mathit{children}_1\}\ ]] \\
\\
\mathit{insert_method} & : & \mathit{m_entry}[[\ \mathit{id}_1.\mathit{id}_2\ ()\ ]] \rightarrow \\
 & & \mathit{transient}(\\
 & & \qquad\qquad \mathit{mt}[[\mathit{id}_3.\mathit{id}_2\ \#\ \mathit{addr}_2\ \mathit{mt}_2]] \rightarrow \mathit{mt}[[\mathit{id}_1.\mathit{id}_2\ \#\ \mathit{addr}_2\ \mathit{mt}_2]] \\
 & & \qquad <\!\!+\ \mathit{mt}[[\mathit{id}_3.\mathit{id}_4\ \#\ \mathit{addr}_2]] \rightarrow \mathit{mt}[[\mathit{id}_3.\mathit{id}_4\ \#\ \mathit{addr}_2\ \mathit{id}_1.\mathit{id}_2\ \#\ \mathit{addr}_2 + 1]] \\
 & & \qquad <\!\!+\ \mathit{mt}[[\ ]] \rightarrow \mathit{mt}[[\mathit{id}_0.\mathit{id}_1\ \#\ 0]] \\
 & &)
\end{array}
$$

Figure 22: Method table construction

Figure 22 shows how the method table construction strategy described can be implemented in TL. When applied to a method declaration, the strategy *insert_method* creates a transient strategy that, though the use of the left-biased choice combinator, captures the three ways a method can be added to a method table. The rule

$$mt[[id_3.id_2\ \#\ addr_2\ mt_2]] \rightarrow mt[[id_1.id_2\ \#\ addr_2\ mt_2]]$$

accounts for the case where a local method definition overwrites an inherited method definition. In this case, id_3 denotes the most recent ancestor where the method id_2 has been declared and id_1 denotes the name of the current class in which the method is being redeclared. The rule

$$mt[[id_3.id_4\ \#\ addr_2]] \rightarrow mt[[id_3.id_4\ \#\ addr_2\ id_1.id_2\ \#\ addr_2 + 1]]$$

accounts for the case where a previously unseen method is declared and must therefore be added to the end of the method table with an offset of $addr_2 + 1$. And finally, the rule

$$mt[[\ ]] \rightarrow mt[[id_0.id_1\ \#\ 0]]$$

accounts for the somewhat special case where a method is added to an empty method table. In this case the offset address is set to 0. Notice that the aggregation of the above rules needs to be encapsulated within a *transient* in order to assure that a method m will only be added to a method table once. For example, it would be incorrect to overwrite an existing method and also add a new (i.e., duplicate) entry to the end of the same method table.

Within the strategy *add_methods*, the evaluation of the strategic expression *seq_tdl insert_method* $method_list_1$ will result in the creation of a method table insertion strategy for each method in $method_list_1$, which is the list containing the methods that are declared in $class_1$. The insertion strategies are sequentially composed by *seq_tdl* and the resulting strategy is ready to be applied to a method table. Let s denote this strategy. The trick that needs to be worked out now is how to apply this value of s to the method table in $class_1$ as well as the method tables of every class which is a descendant of $class_1$. The problem is that s contains transient strategies and the moment a transient strategy applies the value

of s will be forever changed. For example, adding an entry to the method table of $class_1$ will change s so that this element cannot in the future be added to the method tables of any of the descendants of $class_1$. To solve this problem what we need is some way of making a **copy** of the current value of s (i.e., the value before any transient strategies have been applied). The first-order generic traversal TDL_B does just that! In general, the evaluation of a strategic expression of the form $TDL_B\ s\ t$ will perform a top-down left-to-right traversal over the term t and do the following: First, s is applied to the current term t producing a (possibly) new term t' and a (possibly) new strategy s'. Next, a *copy* of the strategy s' is applied to each of the children of t', at which point the process repeats until the entire tree is traversed. As a result the evaluation of the strategic expression

$$TDL_B(seq_tdl\ insert_method\ method_list_1)\ class_1$$

will correctly insert the methods declared in $method_list_1$ into the method table of $class_1$ as well as in the method tables of all the descendants of $class_1$. And finally, the strategy *mt_construction*, when applied to an application app_1 that is in the form of an inheritance tree, will create the proper method tables for each class in app_1. It accomplishes this by traversing app_1 in a top-down left-to-right fashion and applying the strategy *add_methods* to every inheritance tree encountered.

4.5 Inter-class Absolute Address and Offset Address Distribution in TL

At this point, all data indexes within the application have been resolved to symbolic references, all static fields in the application have been assigned absolute addresses in D_{HEAP}, all instance fields in the application have been assigned offset addresses in D_{OBJ}, and all methods in the application have been assigned method table offset addresses in D_{MT}. Given these preconditions, *inter-class absolute address and offset address distribution* concerns itself with the distribution of the aforementioned address values to their corresponding symbolic references in constant pool entries. Note that such references are not particularly constrained by inheritance chains. That is, a class file belonging to one inheritance chain may reference static fields, instance fields, and methods declared in a class file belonging to another inheritance chain.

Figure 23 shows a TL implementation of *inter-class absolute address and offset address distribution.* The evaluation of the strategic expression *lcond_tdl collect_ifields* app_1 will create an instance of the rule

$$c_entry[[\ (index_1,key_1)\ ]] \rightarrow c_entry[[\ (index_1,key_1\ :\ addr_1)\ ]]$$

for each instance field in the application app_1. This rule adds the offset associated with the instance field key_1) to a constant pool entry containing the symbolic reference key_1). These rule instances are composed using the left-biased choice combinator and the resulting strategy is then applied to app_1 using the generic traversal TDL. The effect is that all constant pool entries containing symbolic references to instance fields will be updated so they also contain the corresponding offset address for that instance field. The result of this transformation is then bound to app_2 via a match equation. Next the absolute addresses for all static fields is app_2 is distributed by the same mechanism that was used to distributed instance field offsets. The result is then bound to app_3 via a match equation. And finally, method table offsets are distributed, completing the class loader core as defined in Section 1.2.

distribute_all	:	$app_1 \rightarrow app_4$
	if	$app_2 \ll TDL$ (*lcond_tdl collect_ifields* app_1) app_1
	$\wedge$	$app_3 \ll TDL$ (*lcond_tdl collect_sfields* app_1) app_2
	$\wedge$	$app_4 \ll TDL$ (*lcond_tdl collect_methods* app_1) app_3
collect_ifields	:	$ifield[[key_1 : addr_1]] \rightarrow c_entry[[\ (index_1,key_1)\ ]] \rightarrow c_entry[[\ (index_1,key_1 : addr_1)\ ]]$
collect_sfields	:	$sfield[[key_1 \ @\ addr_1]] \rightarrow c_entry[[\ (index_1,key_1)\ ]] \rightarrow c_entry[[\ (index_1,key_1 \ @\ addr_1)\ ]]$
collect_methods	:	$mt_entry[[key_1 \ \#\ addr_1]] \rightarrow c_entry[[\ (index_1,key_1)\ ]] \rightarrow c_entry[[\ (index_1,key_1 \ \#\ addr_1)\ ]]$

Figure 23: Inter-class distribution

5 Verification and Validation

One motivating factor in the development of TL in general and the SSP specifically is the attainment of strong evidence of correctness. In this section we present current efforts to construct a framework for proving the correctness of TL transformations using the automated theorem prover ACL2.

The goal with respect to the SSP class loader is to prove formally and automatically that the TL implementation of the class loader preserves the properties identified in Section 1.2. To this end, we are beginning to model the behaviors of strategies, traversals, combinators, and conditional rewrites in ACL2. The work described here is only the initial steps towards proving the correctness of the TL implementation of the class loader core. The long-range goal of this work is to reason not only about specific transformations, but also about the internal mechanisms of TL and transformation systems in general. ACL2 has the capability to reason about combinators and traversal strategies, and this reasoning can be reused in other applications that utilize transformation-oriented programming.

5.1 The ACL2 Theorem Prover

ACL2 [8] [9] is a first order, quantifier-free mathematical logic based on recursively defined total functions. It is also a programming language based on the applicative subset of Common Lisp in which users can build executable models of software systems and prove that these models have certain properties. To use ACL2, a user first builds an executable model by writing functions in the ACL2 language. Before a function definition is accepted, ACL2 must prove that the function is total, i.e., it eventually terminates for any input. Once a function is accepted, a user may execute the function to test it or prove theorems about the function.

Theorems are specified using the construct *defthm*. While ACL2's theorem prover is fully automatic, it is usually necessary for a user to supply lemmas to guide the proof. Once a theorem has been proved, it can be used in subsequent proofs. Theorem conjectures have the following general form:

```
(defthm theorem-name
  (implies cond1
           (equal expression1 expression2))
    :rule-classes :rewrite))
```

To prove theorems, ACL2 uses six proof techniques: simplification, elimination of destructors, use of equivalences, generalization, elimination of irrelevance, and induction. The two most important techniques are simplification and induction. After proving a theorem, ACL2 will store the result as a rewrite and use it during simplification. In this case, whenever ACL2 encounters $expression1$, it will try to rewrite $expression1$ to $expression2$ given that $cond1$ is true.

To prove a theorem, $T(n)$, by classical induction, we have to show as a base case that $T(0)$ holds, then as the induction step that $T(n) \rightarrow T(n+1)$. The induction used in ACL2 restates the classical one: to prove $T(n)$ we have to show that $T(0)$ holds, then as the induction step we have to show that $(n \neq 0)$ and $T(n-1) \rightarrow T(n)$. Thus, ACL2 attempts to prove a theorem $T(n)$ by induction from a smaller instances of the theorem, $T(n-1)$.

5.2 Modelling TL in ACL2

We are beginning to model the semantics of TL in ACL2. Our approach is to model the class loader as a function that takes a Java application (set of Java class files) as input and produces a ROM image as an output. The goal is to show the equivalence of these two representations. To prove the equivalence, we create a function that maps a Java application APP to a semantic expression E_{APP}. A second semantic function takes a ROM image and maps it to a semantic expression E_{ROM}. Our conjecture is that E_{APP} is equivalent to E_{ROM} modulo class loading. Rather than show this equivalence for some finite number of test cases, we want to prove that equivalence holds for any valid Java application. We begin by assuming that the Java application and the transformation rules have corresponding tree representations that are inputs to the TL model. The model supports the combinators *FIX_TDL*, *TDL*, and *transient*; we are currently working on supporting the remaining combinators.

ACL2 requires that functions be total. It is therefore impossible to directly model some computational systems. For example, it is not possible to directly model the Java virtual machine since it is possible to write Java programs that do not terminate. The approach to modelling the TL in ACL2 is based on the approach taken to model possibly non-terminating systems [12] [17] [13] [14]. In this approach, a system is described in terms of *states* and a *state transition function*. (This approach has been used to verify parts of implementations of the Java Virtual Machine [12] [17][13].)

A state transition function takes a state as input and returns a new state. The new state is derived from the input state by advancing the system by some number of atomic steps. For example, in the Java virtual machine models, a single step is the execution of a single byte code.

We define a *step* in the ACL2 model of TL to be the application of a single conditional rewrite rule to a specific node in a tree. Our transition function has two inputs: an input state and a number representing how many steps should be taken to produce the output state. Recall that the application of a first-order strategy to a tree generally defines a number of rewrites on that tree. Thus, given a proper number of steps, our transition function implements a strategy.

A state of TL is modeled as a tuple $<CF_I, T, C, TP, CFP, H>$ where CF_I is an input class file; T is a set of rewrite rules; C is a control strategy that controls the application of the transformation rules to the class file; TP is a rewrite pointer that keeps track of the next rewrite rule to be applied; CFP is a Class file pointer that keeps track of the class file entry to which the current rewrite rule will be applied; and a halt flag, H, to show when an execution session ends.

The SSP class loader can be viewed as a function that takes Java application app_0 as input and produces a ROM image CF_{ROM} as output. In Section 2 this core functionality has been decomposed into a sequence of canonical forms. The five canonical forms correspond to index resolution, static field address calculation, instance field offset calculation, method table construction, inter-class absolute address and offset address distribution.

Given a set of state transition functions STF_{form_n} that convert intermediate applications from one canonical form to another, the sequence of intermediates is given below:

$Form_1 = STF_{form_1}(CF_0, nstep_1)$	Index Resolution
$Form_2 = STF_{form_2}(Form_1, nstep_2)$	Static Field Address Calculation
$Form_3 = STF_{form_3}(Form_2, nstep_3)$	Instance Field Offset Calculation
$Form_4 = STF_{form_4}(Form_3, nstep_4)$	Method Table Construction
$CF_{ROM} = STF_{form_5}(Form_4, nstep_5)$	Inter-class Absolute Address and Offset Address Distribution

Here STF_{form_1} represents the transition function that given the correct number of steps $nstep_1$ converts an input class file CF_0 into the first canonical form $Form_1$. Similarly, STF_{form_2} converts a file from the first to the second canonical form. Each transition function will apply a set of rewrite rules to the state according to the specified control strategy. After the appropriate number of steps have been executed, the halt flag, H, is set. The resulting application is stored. This application represents either the resulting ROM image or one of the intermediate forms.

5.3 Verification of the Correctness of the Transformation Rules

In this section, we sketch the proof of the correctness of the rewrite rules that embody the functionality of the SSP class loader. A crucial part of the verification effort is to define a semantic function that determines the equivalence of the input and the output of each transition function. Therefore, for each transition function, there is a semantic function, $SemanticEquiv_n$. Our main conjecture is as follows:

$$\forall(CF_0)SemanticEquiv(CF_0) = SemanticEquiv(STF_{form*}(CF_0), nstep*),$$

which can be proved using transitivity of the following sequence of conjectures:

$$\begin{aligned}\forall(CF_0)SemanticEquiv(CF_0) &= SemanticEquiv_1(STF_{form_1}(CF_0, nstep_1)) \\ &= SemanticEquiv_2(STF_{form_2}(STF_{form_1}(CF_0, nstep_1)), nstep_2) \\ &\ldots \\ &= SemanticEquiv_n(STF_{form*}(CF_0, nstep*))\end{aligned}$$

where STF_{form*} is the composition of the individual transition functions and $number*$ is the total number of steps needed in the transformation of CF_0 to CF_{ROM}. This allows the proof to be constructed incrementally, and therefore, reduces the complexity of the proof.

The goal of the transition function STF_{form_1} is to remove all indirection in the constant pool sections of the class files in CF_0, i.e., perform index resolution. We will also define the accompanied $SematicEquiv_1$ function. Our goal in the example is to prove the following conjecture:

$$SemanticEquiv(CF_0) = SemanticEquiv_1(STF_{form_1}(CF_0, nstep_1))$$

In the proofs shown below, we present the $SemanticEquiv_1$ function. We prove that this function is correct, i.e., it has some desired property. Then we show the conjecture that is required to prove the TL implementation computes an equivalent value.

The goal in this example is to establish a transformational environment that is capable of resolving the pointers in the second column of the table, i.e., to replace each index with the string to which it points. The model shown here is greatly simplified from a true SSP class file. In our current ACL2 model, we represent the constant pool as a list of pairs. Each pair has an index and a value. This format illustrates the notion of indirection within the constant pool. As shown below, the first entry has index 1, and its value is the string "Hello".

```
CP = ((1 "Hello")
      (2 "World")
      (3 2)
      (4 3))
```

Each CP entry consists of two components: the first is a natural number, and the second is either a string (Utf8) or a natural number that is smaller than the first component. In this case, the second component is a pointer to some previous entry in the pool. We require that such references be strictly decreasing, that is, such indirection must point to an earlier entry in the table. This simplifies the proof of termination of the function that follows the resolution chain.

To resolve CP, a rule is needed for each entry of CP that contains a pair $(i\ j)$. This rule relates i and j in a specific context. A second-order transformation rule simplifies the resolution of CP. The rule $TR1 = (i\ j) \rightarrow (x\ i) \rightarrow (x\ j)$ generates as set of rules that transform CP. $TR1$ states that for each entry in the table, create a new first-order rule that replaces the right-hand side of the pair. The generated rules for CP are:

```
TR1.0 = (x 1) → (x "Hello")
TR1.1 = (x 2) → (x "World")
TR1.2 = (x 3) → (x 2)
TR1.3 = (x 4) → (x 3)
```

Rule $TR1.3$, for example, says to replace any entry in the table that has a right-hand value of 4 with an entry that has the same left-hand value, but the right-hand value is set to 3.

The use of the TDL strategy is not enough to resolve all the pointers in CP. Applying all four of the transformation rules to the table once leaves the final table entry as (4 2). Thus, the TDL strategy must be applied as many times as necessary to have all the pointers in the table resolved. This is the FIX_TDL strategy.

The function $fixStrategy$ simulates the behavior of the FIX_TDL strategy in TL. This function takes as input a class file and generates dynamically the corresponding transformation rules. Thereafter, the transformation rules are exhaustively applied to the class file until all the pointers are resolved. The

function $fixStrategy$ achieves this by calling the function that implements the TDL strategy repeatedly until there are no more pointers to resolve.

To verify the transformation rules, we define a semantic function, $resolveLinks$, which copies a table replacing the second component of every entry by the (Utf8) string at the end of the pointer chain. This function uses the function ($getConstant\ n\ cp$), which takes a natural number n and a constant pool table cp. n is an index that is followed until a (Utf8) string is reached. The definition of the function $getConstant$ in ACL2 is as follows.

```
(defun getConstant (n cp)
  (let ((temp (assoc n cp)))
    (cond ((null temp) nil)              ; return nil if no entry
          ((stringp (cadr temp)) (cadr temp))  ; done if utf8
          ((or (not (natp n))            ; nil if table is broken
               (not (natp (cadr temp)))
               (<= n (cadr temp)))
           nil)
          (t (getConstant (cadr temp) cp)))))
```

The function $resolveLinks$ calls the function $resolveLinks1$, which iterates down CP replacing each second component by its associated string value from the original table.

```
(defun resolveLinks1 (tail cp)
  (cond ((endp tail) nil)
        (t (cons (list (car (car tail))
                       (getConstant (car (car tail)) cp))
                 (resolveLinks1 (cdr tail) cp)))))

(defun resolveLinks (cp) (resolveLinks1 cp cp))
```

The following theorem verifies the function $resolveLinks$:

```
(defthm getConstantResolveLinks
  (implies (and (natp n)
                (alistp cp))
           (equal (getConstant n (resolveLinks cp))
                  (getConstant n cp))))
```

ACL2 is not able to prove this theorem directly. In order to prove the theorem, it is first necessary to show that the function $resolveLinks1$ computes the same thing that $resolveLinks1$ does. The following lemma states that if we are given an index (a natural number n) and a table (the association list $tail$), then looking up the index in a resolved table is the same as extracting the constant from the original table.

```
(defthm assocResolveLinks1
  (implies (and (natp n)
                (alistp tail))
           (equal (assoc n (resolveLinks1 tail cp))
                  (if (assoc n tail)
                      (list n (getConstant n cp))
                    nil))))
```

An association list is a list of pairs. The first element of the pair is a key, and the second is the value. The function *assoc* takes a key and an association list and returns the first pair that matches the key. To prove $assocResolveLinks1$, ACL2 chooses an induction schema suggested by the definition of *assoc*. The essential part of the *assoc* function is given below.

```
 (DEFUN ASSOC (X ALIST)
             (COND ((ENDP ALIST) NIL)
                    ((EQL X (CAR (CAR ALIST))) (CAR ALIST))
                    (T (ASSOC X (CDR ALIST)))))
```

The induction has two base cases and one induction step. The base case correspond to the conditions $(ENDP\ ALIST)$ and $(EQL\ X\ (CAR\ (CAR\ ALIST)))$ in the *assoc* definition. Either of these cases will cause the function to halt. The proofs of the base cases are trivial.

The induction step corresponds to the last condition in the *assoc* definition. This condition leads to the recursive call of *assoc*. Let $F(cp, n, tail)$ denotes the main conjecture, i.e.,

```
(implies (and (natp n)
              (alistp tail))
           (equal (assoc n (resolveLinks1 tail cp))
                  (if (assoc n tail)
                      (list n (getConstant n cp))
                    nil))))
```

The induction step can be written as

```
(implies (and (not (endp tail))
              (not (eql n (caar tail)))
              (F cp n (cdr tail)))
     (F cp n tail))
```

ACL2 proves this by simplification appealing to the definitions. With this lemma, ACL2 is able to prove the theorem $getConstantResolveLinks$.

At this point, we have proved the correctness of the semantic function $resolveLinks$. The next step is to use $resolveLinks$ to prove that the semantic of CP is preserved after applying the function STF_{form_1} to it. Our main conjecture is as follows:

```
(defthm resolveLinks-using-trnsf-is-resolveLinks
    (implies
       (and (natp n)
            (alistp cp))
       (equal (getConstant n (fixStrategy cp))
              (getConstant n (resolveLinks cp)))))
```

This theorem claims if *fixStrategy* and *resolveLinks* are given the same constant pool table, then the results of following the resolution chains for an index in both tables are identical. While this result may seem trivial, it provides significant insight into how this approach can be used to verify significant applications written in TL.

6 Related Work

In this section we discuss related work in the areas of rewriting/strategic programming as well as the verification style we are pursuing.

6.1 Rewriting and Strategic Programming

There are a number of rewriting and strategic systems where a rewrite-based implementation of the class loader core could be considered. Among these systems are ELAN [2], Stratego [22], and ASF+SDF [1]. One of the unique features of TL is its use of higher-order strategies as the mechanism to aggregate data (e.g., all indexes in a constant pool and the Utf8 data to which they resolve) and first-order strategy application as the mechanism to distribute data (e.g., index resolution) throughout a term structure (e.g., a class file or application). In contrast, both ELAN and ASF+SDF [5], use parameterization to collect and distribute data throughout a term structure. That is, parameters are added to rewrite rules which can then be passed down and applied at various points within a term structure. In a parameter-based approach, aggregations of data (e.g., all indexes in the constant pool) are typically converted into an internal representation such as a list and must be accompanied with an associated lookup function.

In contrast to parameterization, Stratego supports an approach that arguably can be considered the first-order cousin of higher-order strategy construction mechanisms of TL. Stratego is a first-order strategic programming system that has two constructs related to the higher-order strategies presented in the paper: *contextual rules* and *scoped dynamic rewrite rules*. In [21], contextual rules are used to distribute data within a term structure and can be seen as a first-order cousin of the higher-order rules presented in this paper.

In [20], an approach to the distributed data problem is taken that is similar to what we have described. Here the distributed data problem is viewed from a context-free/context-sensitive perspective. In particular, semantic relationships between portions of a term are seen as representing context-sensitive relationships. Dynamic rewrite rules are developed as a mechanism for capturing context-sensitive relationships between portions of a term. Dynamic rewrite rules are named rewrite rules that can be instantiated at runtime (i.e., dynamically) yielding a rule instance which is then added to the existing rule base. Similar to the higher-order approach taken by TL, in Stratego the program itself is the driver

behind the instantiation of dynamic rule variables. The lifetime of dynamic rules can be explicitly constrained in strategy definitions by the scoping operator {| ... |}.

Primary differences between our approach and the scoped dynamic rules described in [20] are the following:

1. In our approach, we view the rule base as a *strategy* that is created dynamically. The $\oplus$ combinator provides the user explicit control over the structure of this strategy.

2. Though the transient combinator has no direct analogy within scoped dynamic rewrite rules, its effects can be simulated in Stratego [23]. However, it is somewhat unclear whether a single approach/method can be used in Stratego to simulate all the behaviors resulting from the interaction between higher-order strategies and transients.

3. The *hide* combinator has no analogy in Stratego.

6.2 Verification

State transition approach was investigated in [12] [17] [13] [14] In [4], Boyer and Yu used Nqthm [3], the predecessor of ACL2, to formalize a substantial subset of a commercial microprocessor, the Motorola MC68020 [M85]. 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. In [14], Moore 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 p_0 you obtain state p_n by running piton forward n steps. However, the alternative approach is to map p_0 to down to a FM8502 state (or core image), through a function defined in ACL2, 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 Piton was mechanically proved correct. In [12] [17][13] ACL2 has been used in verifying the JVM by analyzing the bytecode produced for it. 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

From a conceptual standpoint, we believe that transformation provides a natural framework in which the functionality of the class loader core can be considered. However, the intricacy of data interactions as well as the structural complexity of Java class files presents a number of challenges to traditional rewriting and strategic frameworks. Foremost among these challenges is the treatment of term-specific data and its distribution throughout a term structure. Though table construction and parameterization

are techniques capable of realizing data distribution, their use departs from rewriting in its purest sense. Our research is based on the premise that higher-order rewriting provides a mechanism for dealing with the treatment and distribution of term-specific data conforming to the tenets of rewriting. In a higher-order framework, the use of such data is expressed as a rule. Instantiation of such rules can be done using standard (albeit higher-order) mechanisms controlling rule application (e.g., traversal). Typically, a traversal-driven application of a higher-order rule will result in a number of instantiations. If left unstructured, these instantiations can be collectively seen as constituting a rule base whose creation takes place dynamically. However, such rule bases again encounter difficulties with respect to confluence and termination. In order to address this concern the notion of strategy construction is lifted to the higher-order as well. That is, instantiations result in rule bases that are structured to form strategies. Nevertheless, in many cases, simply lifting first-order control mechanisms to the higher-order does not permit the construction of strategies that are sufficiently refined. This difficulty is alleviated though the introduction of the *transient* and *hide* combinators. The interplay between these combinators, higher-order rules, and more traditional control mechanisms enables a the functionality of the class loader core to be concisely expressed. In spite of this, reasoning about the correctness of higher-order strategies is conceptually somewhat of a departure from the reasoning used when considering first-order rewrite rules. Our current efforts in using ACL2 reflects our initial efforts in formalizing our reasoning process in an automatable fashion. This effort involves mapping our approach to reasoning about TL strategies onto proven approaches to reasoning about software.

References

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

[2] 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.

[3] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, NY, 1988.

[4] R. S. Boyer and Y. Yu. *Automated proofs of object code for a widely used microprocessor*, Journal of the ACM, Vol. 43, No. 1, January 1996, pp. 166-192.

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

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

[8] M. Kaufmann, P. Manolios, et al., *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June 2000.

[9] M. Kaufmann, and P. Manolios, et al., editors. *Computer-Aided Reasoning: Case Studies*, Kluwer Academic Publishers, June 2000.

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

[11] 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.

[12] 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, 2003, pp. 227-290.

[13] J S Moore, *Proving Theorems about Java-like Byte Code*, in E.-R. Olderog and B. Steffen (eds.) Correct System Design – Recent Insights and Advances, LNCS 1710, 1999, pp. 139-162.

[14] J S Moore, Piton: A Mechanically Verified Assembly- Level Language, Automated Reasoning Series, Kluwer Academic Publishers, 1996.

[15] J S Moore and B. Brock et al., *ACL2 Theorems about Commercial Microprocessors*, In M. Srivas and A. Camilleri (eds.) Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96), Springer-Verlag, 1996, pp. 275-293.

[16] J S Moore and T. Lynch, *A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5k86 Floating- Point Division Program*, IEEE Transactions on Computers, Vol. 47, No. 9, September1998, pp. 913-926.

[17] J S Moore and G. Porter, *An Executable Formal Java Virtual Machine Thread Model*, in Java Virtual Machine Research and Technology Symposium (JVM '01), USENIX, April, 2001.

[18] D. Russinoff, *A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating Point Multiplication, Division and Square Root Instructions*, 1106 W. 9th St., Austin TX 78703, July 1998.

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

[20] 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.

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

[22] E. Visser, Z. Benaissa, and A. Tolmach. *Building Program Optimizers with Rewriting Strategies.* Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98).

[23] E. Visser. Personal communication, Feb. 18, 2004.

[24] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. L. Winter. *The SSP: An Example of High-Assurance System Engineering.* The 8th IEEE International Symposium on High Assurance Systems Engineering (HASE'2004), March 2004.

[25] V.L. Winter, S. Roach, G. Wickstrom. *Transformation-Oriented Programming: A Development Methodology for High Assurance Software.* Advances in Computers vol 58.

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

[27] V.L. Winter. *Strategy Construction in the Higher-Order Framework of TL.* The 5th International Workshop on Rule-Based Programming (RULE 2004) (to appear).

[28] V.L. Winter. *Strategy Application, Observability, and the Choice Combinator.* Sandia Technical Report.

[29] V. L. Winter. *The Observation of Choice.* The Journal of Logic and Algebraic Programming (submitted).

Distribution:

1	MS0510	Greg Wickstrom, 2116
1	MS0510	James McCoy, 2116
1	MS0510	Anna Schauer, 2116
3		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