

A Transformational Overview of the Core Functionality of an Abstract Class Loader for the SSP *

Victor L. Winter,[†] Jason Beranek, and Azamatbek Mametjanov

University of Nebraska at Omaha

Department of Computer Science

{ *vwinter, jberanek, amametjanov* } @mail.unomaha.edu

Fares Fraij and Steve Roach

University of Texas at El Paso

Department of Computer Science

{ *fzfraij, sroach* } @utep.edu

Greg Wickstrom

Sandia National Laboratories

Department of Surety Electronics and Software

glwicks@sandia.gov

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. The functionality of the class loader can be implemented using strategic programming techniques. The incremental nature of strategic programming is amenable to formal verification. This article gives an overview of the core class loading activities for the SSP, their implementation in the strategic programming language TL, and outlines the approach to formal verification of the implementation.

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) [7][12] 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,

*This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

[†]Victor Winter was also partially supported by NSF grant number CCR-0209187.

(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 subset of the Java class file structure needed 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 TL transformations that implement the class loader.

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). A class file image contains a ROM constant pool and a method table and methods section. There are two major differences between ROM constant pools and constant pools found in Java class files. In ROM constant pools, *class*, *field*, and *method* entries are represented in terms of absolute and offset addresses along with corresponding data as opposed to *symbolic references* along with data in Java class files. Secondly, entries in ROM constant pools are limited to *constant integer*, *constant long*, *static field*, *instance field*, *class*, and *method*. In this paper, we restrict our attention only to virtual methods.

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)

1.2. An overview of class loader requirements

There are a number constraints on how object offsets and method table offsets must be calculated. These can be expressed in the form of properties that a resolved collection of encoded symbolic references must possess in order to be correct, including the following:

- **Unique-Offset:** All instance fields must be resolved to unique object offsets.
- **Consistent-Offset:** Instance fields must be resolved to offsets in a manner that is consistent with upcasting.
- **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.
- **Unique-Address:** Within an application, every static field must be resolved to a unique 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. In other words, the information corresponding to all (re)definitions of a particular method must reside in the same position in every method table.

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 encoded symbolic references and addresses and addresses within a physical address space.

In addition to correctness properties, the resolution of encoded symbolic references must also satisfy a number of efficiency-based constraints.

- **Spatial-Efficiency:**

- Instance fields should be packed as tightly as possible in objects and static fields should be tightly packed in the heap.
- Object offset addresses must be in byte units (i.e., 8-bit quantities).

- **Temporal-Efficiency:**

- 64-bit, 32-bit, and 16-bit values should not span respective bit boundaries within the heap or ROM address space. For example, a 32-bit value must begin on a word boundary (i.e., a byte address that is a multiple of four).

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 encoded symbolic references and physical addresses and between addresses within an address space. Figure 1 lists the contexts that must be considered.

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. The following is a high-level definition of the class loader core.

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

2. Overview of the structure of class files

Figure 2 gives a top-level view of the components contained in a Java class file. More information about the structure of these components as well as class files in general are available [6][11].

In this paper we restrict our attention to the following class loading activities as they pertain to the SSP: (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*.

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.

Given the class files shown in Figure 3, we ask "What is the symbolic reference of the index 1 with respect to the constant pool (CP) of the first class file?" 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. For example, in Figure 3 the index 2 will resolve to "A.x1". The remaining indexes are resolved in a similar fashion resulting in resolved indexes in Figure 4. Complete details on the structure of constant pools can be found in the literature [6][11].

Static Field Address Calculation. The goal of static field address calculation is to assign a unique absolute heap 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. For example, in Figure 4 the absolute address assigned to the static field A.x1 is 0. 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.

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 4 shows possible offset calculations for the instance fields of classes A and B. For example, the instance field A.a2 is assigned the offset address 1.

Method Table Construction. Encoded symbolic refer-

ences to methods must ultimately be resolvable to the address where the bytecodes for the method reside. 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. Methods must be resolved in a manner that is consistent within this framework.

A standard solution to the problem is to construct a method table for each class. 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 bytecodes corresponding to the implementation of a method as seen from the perspective of a particular class. In order for the indirection provided by the method table solve our problem, all classes that inherit or redefine a certain function (e.g., *foo*) must store data related to this function in the same relative position (i.e., offset) in their method table. The method table to which this offset is applied depends on the class from which an object is derived.

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 from the class in which the data originates. Thus, 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*). Note that the CP of class A has inter-class references to instance fields of class B and the CP of B has inter-class references to instance fields of class C (class C not shown) in Figures 3 and 4.

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

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

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()

Figure 3. Two 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()

Figure 4. The two abstract class files shown in Figure 3 after class loading

to execution. Under these conditions, the functionality of a class loader is well suited to a transformation-oriented implementation [16]. 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.

3.1. The basic constructs of TL

TL is a higher-order strategic programming language [15][14][13]. 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. For example, $stmt[[id_1 = 5]]$ is a tree with root $stmt$ and leaves $id_1, =,$ and 5 .
2. *rewrite rules* – A *rewrite rule* is a construct, whose basic form is $lhs \rightarrow rhs$, for specifying that the pattern lhs is to be replaced by the pattern rhs . In a strategic system, *rewrite rules* are also considered to be a degenerative form of *strategy*. A rule applies if its left-hand side (i.e., lhs) matches the term to which it is applied.
3. *conditions* – A *condition* is a boolean-valued expression associated with a rewrite rule that restricts the application of the rule. For example $r_1 : stmt_1 \rightarrow stmt[[id_1 = 5]]$ if $stmt_1 \ll stmt[[id_1 = 4 + 1]]$

is a conditional rule labeled r_1 transforming $stmt_1$ to $stmt[[id_1 = 5]]$ provided the *condition* that $stmt_1$ is a tree of the form $stmt[[id_1 = 4 + 1]]$ is satisfied.

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*. Three combinators are provided: (1) sequential composition ($;$), (2) left-biased composition ($<+$), (3) right-biased composition ($+>$). Let s_1 and s_2 denote two strategies. The expression $s_1; s_2$ denotes the sequential composition of s_1 and s_2 . When applied to a term t this strategy will first apply s_1 to t and then apply s_2 to the result. In contrast, the expression $s_1 <+ s_2$ denotes the left-biased composition of s_1 and s_2 . When applied to a term t the application of s_1 to t is attempted and if that succeeds the result is returned, otherwise the result of the application of s_2 to t is returned. The expression $s_2 +> s_1$ is equivalent to $s_1 <+ s_2$. TL also has two unary combinators unique to it; the *transient* and *hide* combinators which are discussed in Section 4.2.
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. An example of a generic traversal is *TDL* which performs a top-down left-to-right traversal of a term.
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.
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 $n - 1$.

More detailed descriptions of TL constructs are available [15][14][13].

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 with respect to the class file structure described in Figure 5. This implementation has been developed using the HATS system, an IDE for strategic programming supporting a restricted version of TL. The HATS system is freely available [3].

4.1. Index resolution in TL

In Figure 6, the strategy named *index_resolution* gives an implementation of index resolution in TL. The behavior of the *index_resolution* strategy is as follows. When applied to a class file structure $class_0$ the strategy *index_resolution* will first evaluate the strategic expression $lcond_tdl\ cp_normalize\ class_0$. Within this expression, the strategy *lcond_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 *lcond_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:

$$c_entry[[(index_1, data_1)]]$$

into a first-order rewrite rule of the form:

$$data[[index_1]] \rightarrow data_1$$

When applied to the entries of the constant pool of $class_0$ a number of instances of the rule $data[[index_1]] \rightarrow data_1$

app	::=	app class ϵ
class	::=	{ class_id parent_id info children }
children	::=	children class ϵ
info	::=	cp fields methods
class_id	::=	data
parent_id	::=	data
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	::=	data . data data address_type addr id index
address_type	::=	@ # :
index	::=	integer
addr	::=	integer
id	::=	ident

Figure 5. An extended-BNF grammar describing a simplified application in terms of a list of class files

will be generated. These rule instances are then composed by *lcond_tdl* using TL's left-biased conditional composition operator $<+$. This composition is part of the semantics of *lcond_tdl* – which conditionally composes the results generated from its tdl traversal. The resulting first-order strategy is of the form $r_1 <+ r_2 <+ \dots r_n$ where r_i is the instance of $data[[index_1]] \rightarrow data_1$ corresponding to the i^{th} constant pool entry. This first-order strategy is then applied to $class_0$ in a top-down left-to-right fashion by the first order generic traversal *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, 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* [15][14] and *hide* [13]. Both of these combinators are unique to TL.

<i>index_resolution</i>	:	$class_0 \rightarrow TDL(lcond_tdl\ cp_normalize\ class_0)\ class_0$
<i>cp_normalize</i>	:	$c_entry[[(index_1 , data_1)]] \rightarrow data[[index_1]] \rightarrow data_1$
<i>static_addresses</i>	:	$app_0 \rightarrow TDL(lcond_tdl\ sfield_sum\ app_0)\ app_0$
<i>sfield_sum</i>	:	$sfield_0 \rightarrow$ $hide(sfield[[data_1 @ addr_1]]) \rightarrow sfield[[data_1 @ addr_1 + 1]]) \leftarrow+ transient(sfield_1 \rightarrow sfield_1)$
<i>instance_offsets</i>	:	$app_1 \rightarrow app_3$ if $app_2 \ll TDL(seq_tdl\ create_hierarchy\ app_1)\ app[[\{ Obj\ Obj \}]]$ $\wedge\ app_3 \ll TDL\ sum_offsets\ app_2$
<i>create_hierarchy</i>	:	$class_1 \rightarrow class[[\{ id_2\ id_3\ info_2\ children_2 \}]]$ if $class_1 \ll class[[\{ id_1\ id_2\ info_1\ children_1 \}]]$
<i>sum_offsets</i>	:	$class_0 \rightarrow TDL(lcond_tdl\ partial_sum\ instance_1)\ class_0$ if $class_0 \ll class[[\{ class_id_1\ parent_id_1\ cp_1\ statics_1\ instance_1\ methods_1\ children_1 \}]]$
<i>partial_sum</i>	:	$instance_0 \rightarrow TDL(lcond_tdl\ local_ifield_sum\ instance_0)\ instance_0$
<i>local_ifield_sum</i>	:	$ifield_0 \rightarrow$ $hide(ifield[[data_1 : addr_1]]) \rightarrow ifield[[data_1 : addr_1 + 1]]) \leftarrow+ transient(ifield_1 \rightarrow ifield_1)$
<i>mt_construction</i>	:	$app_1 \rightarrow TDL\ add_methods\ app_1$
<i>add_methods</i>	:	$class_1 \rightarrow TDL_B(seq_tdl\ insert_method\ method_list_1)\ class_1$ if $class_1 \ll class[[\{ class_id_1\ parent_id_1\ cp_1\ fields_1\ mt_1\ method_list_1\ children_1 \}]]$
<i>insert_method</i>	:	$m_entry[[id_1.id_2\ 0]] \rightarrow$ $transient($ $mt[[id_3.id_2 \# addr_2\ mt_2]] \rightarrow mt[[id_1.id_2 \# addr_2\ mt_2]]$ $\leftarrow+ mt[[id_3.id_4 \# addr_2]] \rightarrow mt[[id_3.id_4 \# addr_2\ id_1.id_2 \# addr_2 + 1]]$ $\leftarrow+ mt[[]] \rightarrow mt[[id_0.id_1 \# 0]]$ $)$
<i>distribute_all</i>	:	$app_1 \rightarrow TDL(lcond_tdl(collect_ifields\ \leftarrow+ collect_sfields\ \leftarrow+ collect_methods)\ app_1)\ app_1$
<i>collect_ifields</i>	:	$ifield[[data_1 : addr_1]] \rightarrow c_entry[[(index_1, data_1)]] \rightarrow c_entry[[(index_1, data_1 : addr_1)]]$
<i>collect_sfields</i>	:	$sfield[[data_1 @ addr_1]] \rightarrow c_entry[[(index_1, data_1)]] \rightarrow c_entry[[(index_1, data_1 @ addr_1)]]$
<i>collect_methods</i>	:	$mt_entry[[data_1 \# addr_1]] \rightarrow c_entry[[(index_1, data_1)]] \rightarrow c_entry[[(index_1, data_1 \# addr_1)]]$

Figure 6. An Abstract Implementation of the class loader core for the SSP

The *transient* combinator can be used to create a strategy that can be applied at most once. 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 $s_1; s_2$. A consequence of this is that it becomes possible to use the *transient* and *hide* combinator to construct strategies in which an increasing number of sub-strategies are applied to a particular class of terms. Let t_1 and t_2 denote two terms to which the strategy s_1 can be successfully applied. The following strategy:

$$transient(s_1) <+ hide(s_1) <+ s_2$$

will applied to t_1 to the above strategy will result in s_1 being applied at which time the transient combinator restricts any future application of $transient(s_1)$. Under these conditions the strategy is now applied to the term t_2 . In this case the sub-strategy $hide(s_1)$ will apply after which s_2 will be applied. In summary, s_1 is applied to t_1 and $s_1; s_2$ is applied to t_2 . In this paper, the *hide* combinator is used by a higher-order strategy to dynamically construct a counter (a sum) that incrementally assigns successive absolute addresses to the static fields within a collection of class files.

In Figure 6, the strategy named *static_addresses* is 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 following increment technique. 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 app₀* will traverse *app₀* in a tdl fashion and apply the strategy *sfield_sum*. This will produce one instance of the strategy

$$hide(sfield[[data_1@addr_1]] \rightarrow sfield[[data_1@addr_1 + 1]]) <+ transient(sfield_1 \rightarrow sfield_1)$$

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 *app₀* using the generic traversal *TDL*. This has the effect of assigning a unique absolute offset to each static field in *app₀*.

4.3. Instance field offset calculation in TL

In Figure 6, the strategy named *instance_offsets* is a TL implementation of instance offset calculation. When applied to *app₁*, the strategy *instance_offsets* will first restructure *app₁* into an inheritance tree. The evaluation of the

strategic expression *seq.tdl create_hierarchy app₁* will perform a top-down left-to-right traversal on *app₁* and apply the higher-order strategy *create_hierarchy* to each class file encountered. The application of *create_hierarchy* to *class₁* will produce a first-order strategy that places *class₁* into the children list of its parent class. Notice that in the strategy *create_hierarchy*, *id₁* and *id₂* respectively denote the class name and parent class name of *class₁*. By definition, the parent class of *class₁* is that class which has *id₂* 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 *class₁* into the children list of its parent class.

For each class in *app₁* 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 *app₂* 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 app₂* will traverse the inheritance tree *app₂* 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 *class₁* 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 *class₁*. This has the effect of (1) assigning proper (i.e., completed) offsets for the instance fields in *class₁*, and (2) assigning partially completed offsets for the instance fields of all the classes which are descendants of *class₁*. 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., when it is treated as the root of an inheritance tree).

4.4. Method table construction in TL

In Figure 6, the strategy named *mt_construction* is a TL implementation of method table construction. 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

$$\begin{aligned} & mt[[id_3.id_4 \# addr_2]] \\ \rightarrow & \\ & mt[[id_3.id_4 \# addr_2 id_1.id_2 \# addr_2 + 1]] \end{aligned}$$

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 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 question now is how to apply 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, we need 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

In Figure 6, the strategy named *distribute_all* is 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

$$\begin{aligned} & c_entry[[(index_1, data_1)]] \\ \rightarrow & \\ & c_entry[[(index_1, data_1 : addr_1)]] \end{aligned}$$

for each instance field in the application app_1 . This rule adds the offset associated with the instance field ($data_1$) to a constant pool entry containing the symbolic reference ($data_1$). 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.

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 briefly describe the verification approach.

5.1. Modelling TL in ACL2

We are beginning to model the semantics of TL in ACL2 [4][5]. ACL2 is an automated inductive theorem prover that has been used to verify a variety of microprocessor implementations [1][9][8][2][10]. 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.

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

$$\langle CF_I, T, C, TP, CFP, H \rangle$$

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.

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 in Figure 7.

In Figure 7, 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.

5.2. Verification of the correctness of the transformation rules

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:

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

which can be proved using transitivity of the sequence of conjectures shown in Figure 8.

In Figure 8, 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.

6. 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

$Form_1$	$= STF_{form_1}(CF_0, nstep_1)$	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

Figure 7. Intermediate forms

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

Figure 8. A sequence of conjectures

of the *transient* and *hide* combinators. The interplay between these combinators, higher-order rules, and more traditional control mechanisms enables 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] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *J. ACM*, 43(1):166–192, 1996.
- [2] B. Brock, M. Kaufmann, and J. S. Moore. ACL2 theorems about commercial microprocessors. In *FMCAD*, pages 275–293, 1996.
- [3] HATS. <http://faculty.ist.unomaha.edu/winter/hats-uno/hatsweb/index.html>.
- [4] M. Kaufmann and P. M. et al., editors. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
- [5] M. Kaufmann and P. M. et al., editors. *Computer-Aided Reasoning: Case Studies*. Kluwer Academic Publishers, 2000.
- [6] T. Lindholm and F. Yellin, editors. *The Java Virtual Machine (Second Edition)*. Addison-Wesley, 1999.
- [7] J. A. McCoy. An embedded system for safe, secure and reliable execution of high consequence software. In *HASE 2004: The 5th IEEE International Symposium on High Assurance Systems Engineering*, 2000.
- [8] J. S. Moore, editor. *Piton: A Mechanically Verified Assembly-Level Language*. Automated Reasoning Series. Kluwer Academic Publishers, 1996.
- [9] J. S. Moore. Proving Theorems about Java and the JVM with ACL2. In *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003.
- [10] J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the amd5_k86tm floating point division program. *IEEE Trans. Computers*, 47(9):913–926, 1998.
- [11] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1998.
- [12] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. L. Winter. The ssp: An example of high-assurance system engineering. In *HASE 2004: The 8th IEEE International Symposium on High Assurance Systems Engineering*, 2004.
- [13] V. Winter. Strategy application, observability, and the choice combinator. Technical Report SAND2004-0871, Sandia National Laboratories, March 2004.
- [14] V. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS) (to appear)*, 2004.
- [15] V. Winter and M. Subramaniam. Dynamic Strategies, Transient Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.
- [16] V. L. Winter, S. Roach, and G. Wickstrom. Transformation-oriented Programming: A development methodology for high assurance software. In M. Zelkowitz, editor, *Advances in Computers: Highly Dependable Software*, volume 58, pages 47 – 116, 2003.