Verifying Transformation Rules of the HATS High-Assurance Transformation System: An Approach

Steve Roach Fares Fraij

Department of Computer Science
The University of Texas at El Paso
{sroach, fares}@cs.utep.edu

Abstract

In high-consequence systems, there is a high cost associated with failure. Thus, there should be strong evidence that the systems will not fail in the field. Such evidence cannot be obtained by traditional verification methods such as testing. *Transformation-oriented programming (TOP)* is a promising formal software development technique. In TOP, a source file that represents a correct formal specification is continuously refined to produce an implementation. The High-Assurance Transformation System (HATS) is an example of TOP that takes as input a source file and a *transformation language program (TLP)* which represents a sequence of actions to be applied to the source file. HATS users express these actions as a sequence of *transformation rules* and *control strategies*. HATS has been used to develop an application for a high-consequence system, namely the Sandia Secure Processor (SSP). The application is the SSP-classloader in which a source file (class file) is incrementally refined through five canonical forms until it becomes a ROM image that will be executed by the SSP hardware.

In this paper, we introduce an approach to proving the correctness of the TLPs that produce the five canonical forms using ACL2. Our goal is to verify that a TLP that produces a canonical form preserves the semantics of a class file. To achieve this goal, a semantic function that describes the behavior of the TLP must be identified. We have successfully built a simplified model of the TLP that produces the first canonical form, developed a semantic function for this model, and proved that the model preserves the semantic of the class file.

1 Introduction

The complexity of software solutions of elaborate, real-life problems is increasing. Even with extensive effort, software solutions may contain errors. Estimates place the number of software defects in newly-written, uncommented code at 50 per 1,000 lines. In spite of thoroughly testing the code, this number remains around ten [RH04]. In high-consequence systems, there is a high cost associated with failure. In addition, if the cost of failure is measured in human life, then the systems are called *safety-critical systems*. In order to certify a high-consequence system, a failure rate of one failure in 10^9 operational hours must be provided [BS93] [WR03]. Such a system is expected to fail no more than once in 114,155 years [WR03]. The strong evidence that the system will not fail in the field cannot be obtained by traditional verification methods such as testing.

Transformation-oriented programming (TOP) [WR03] is a promising formal software development paradigm. In TOP, an input source file that contains a correct formal

specification of a software system is refined by a user-defined transformation sequence in order to produce an implementation. The correctness of a TOP implementation can be argued since the implementation is the result of refining the source file incrementally using a well-defined transformation sequence. However, such an argument is not enough to certify a high-consequence application. Strong evidence should be provided that the transformation sequence is correctness-preserving. Such evidence can be provided by using automatic reasoning systems like ACL2 [KM00a] [KM00b].

The High-Assurance Transformation System (HATS) [WR04A] [WR04B] [WR04C] [WR03] [W99] is an example of a TOP system. HATS provides a Transformation Language Program (TLP) that facilitates the development of transformation rules and strategies, which control the application of these rules. The main two inputs to HATS are: a source file that represents a correct formal specification and a TLP. The TLP consists of a sequence of transformation rules that embodies the functionality of a certain application and control strategies that control the application of the sequence to the source file. Then, HATS engine will refine the source file using the TLP to produce an implementation.

HATS was used to implement part of the Sandia Secure Processor (SSP) [WR03][WR04B], an application for high-consequence systems. The application program is the SSP-classloader that takes a source file (class file), C_0, as input and produces a ROM image, C_{ROM}, as output. The C_{ROM} then can be executed by the SSP hardware. The classloader can be decomposed into a sequence of five canonical forms in which a class file is transformed from C_0 to C_{ROM}. This decomposition facilitates the verification of the correctness of the SSP-classloader using ACL2.

In this paper, we introduce an approach to proving the correctness of a HATS implementation of the SSP-classloader. We develop models and techniques using ACL2 to prove the correctness of each of the HATS TLPs that represents each one of the canonical forms. We have successfully built a simplified model of the TLP that represents the first canonical form, developed a semantic function for this form, and proved that the model preserves the semantics of the source file. Ultimately, we will show that the TLP is correctness preserving. To our knowledge, no such models and techniques exist.

The organization of this paper is as follows. Section 2 introduces the High-Assurance Transformation System (HATS). It describes the architecture of HATS, and the syntax and semantics of the transformation rules and control strategies that HATS support. It also presents the Program Transformation Language (PTL) for defining transformation rules: first- and second-order transformation rules, control strategies, and combinators. This section ends with an example that shows the concept of transformation rules and control strategies. Section 3 discusses the motivating high-consequence system, namely the SSP, its relation to the Java Virtual Machine [LY99] (JVM), and its two main components: the SSP-classloader and the SSP-runtime. Section 4 investigates the simplified ACL2 specifications of the TLP that represents the first canonical form of the SSP-classloader and presents the verification of that part. Section 5 concludes this paper and discusses future directions.

2 HATS

HATS is a transformation system that accepts two inputs, a source file in a specification language and a TLP. HATS produces a target program in an implementation language. Figure 2-1 illustrates this idea. The TLP specifies transformation rules and control strategies, which affect how the transformation rules are applied to the source file. The programs in HATS are written in a context-free grammar and are stored internally as syntax derivation trees (SDTs).

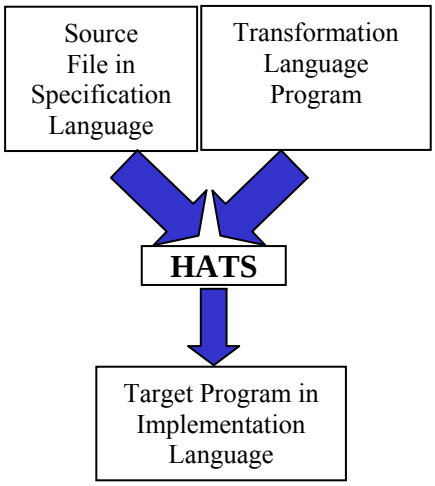

Figure 2-1: HATS overview

2.1 Architecture

HATS consists of five main components that are shown in Figure 2-2. The first component is a *target parser* that is able to identify domain elements in the source file. The second is a program parser that can read TLPs. The third is the *HATS rewriting engine*, which applies the TLP to the source file. The fourth is a *pretty printer* that formats the results in human-readable form. Finally, the *GUI* contains a debugger that facilitates the comprehension of the transformation rules and how they are being applied to a particular part of a source file.

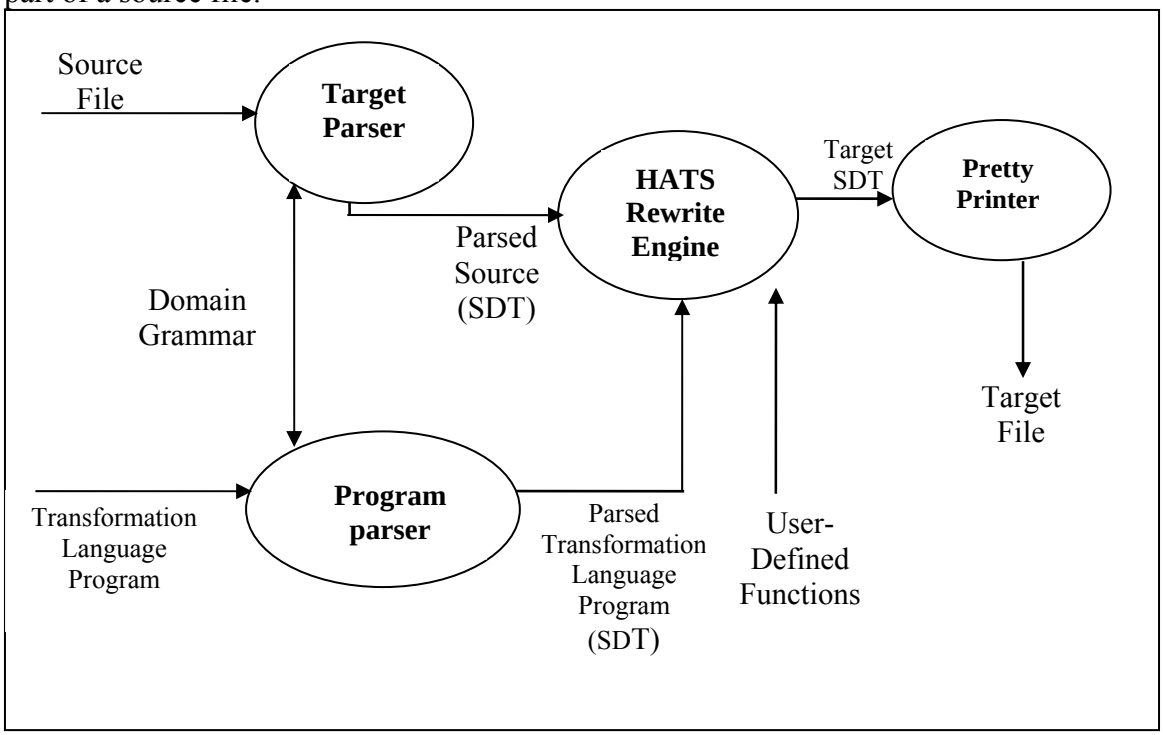

Figure 2-2: HATS data flow diagram

The parser accepts an input string, which represents the source file, and reads it to generate an SDT, which is described by a domain grammar. The resulting SDT will be the destination of the transformation rules. In this work, “SDT”, “parse tree”, and “terms”

are synonymous. The TLP is also converted to an SDT. Then, the rewriter attempts to apply the transformation rules and strategies in the TLP to the source SDT in an attempt to transform it into a new SDT that represents the target program. Finally, the pretty printer is used to display the results.

2.2 Transformation Language Program

HATS provides a special purpose language for defining TLPs. A TLP contains primitives such as types, operations, identifiers, constants, and comments. It also contains language constructs. The important primitives are the ones used to define transformation rules and control strategies. A TLP may also contain combinators, which define how a sequence of transformation rules will be manipulated. A simplified syntax will be used in this work.

The structure of a TLP is shown in Figure 2-3. It consists of four main sections. The first section is the global declaration section. The second is the transform function section, which defines the transformation rules that will be used to convert a source file into target program. The third section is the local declaration section, which usually contains the declaration of the target program and the result as an SDT type. The last section is the *main body* of the program, which specifies the control strategy that will be used to apply transformation rules to the target program SDT and prints out the result.

(a) Transformation program structure	(b) Transformation program fragment
global declaration section	_____
transform function section	transform rule_name = [$rule_1$ @$rule_2$ ... @$rule_n$]
local declaration section	SDT: input_program, result

main body of program	input_program := input("target-program");
	result := fix(post_order, input_program, rule_name);
	output_program_string(result).

Figure 2-3: The recommended structure for a transformation program

2.2.1 Transformation Rules

A transformation rule consists of a left-hand side, that represents a term to transform, a right-hand side, that represents the new term to replace the old one, and a rewrite operator, denoted by the symbol →, used to connect the left-hand side with the right-hand side. Thus, a rewrite rule has the following general form.

left-hand side → *right-hand side*

The left-hand side may contain variables. As an example,

$$natp\ (n) \rightarrow IntLessThanOrEqual\ (0,\ n),$$

is a rewrite rule where *n* is a variable, *0* is a constant that represents the natural number zero, and *natp* and *IntLessThanOrEqual* are predicates. The *natp* predicate returns true if its argument is a natural number. The rewrite rule states that *natp* can be implemented using the *IntLessThanOrEqual* predicate with its first argument set to the constant 0.
The transformation proceeds as follows: for a given term *t*, HATS determines whether it is an instance of the left-hand side of a transformation rule. If it is, then *t* will be matched with the left-hand side of that rule producing a substitution list such that *t* and the left-hand side of the rule are identical. This process is called *matching*. The substitution list is used to instantiate the right-hand side of the rule, which replaces *t*. For instance, suppose that *t* is the term *natp (10).* This term will be matched with the left-hand side of the rule, namely *natp (n).* As a result, *n* will be bound to the constant *10*, and the substitution list is *{n/10}*. The right-hand side of the rule is instantiated using this substitution, and the result is *IntLessThanOrEqual (0, 10)*. This term will replace *natp (10)*. Thus, the result of applying *natp (n) → IntLessThanOrEqual (0, n)* to *natp (10)* is *IntLessThanOrEqual (0, 10).*

$$\frac{natp\ (n) \rightarrow IntLessThanOrEqual\ (0,\ n) \quad natp\ (10)}{IntLessThanOrEqual\ (0,\ 10)}$$

Another variation of transformation rules is the conditional transformation rule. In this type, the transformation rules are annotated with conditions. The general form of such rules is as follows.

$$left\text{-}hand\ side \rightarrow\ right\text{-}hand\ side\ if\ C,$$

where *C* denotes a Boolean formula. In conditional transformation rule, a term *t* is matched with the left-hand side of a transformation rule and a substitution list is constructed as a result of the matching process. This substitution list is applied to the condition *C*, and *C* is evaluated. If *C* is true, then the right-hand side of the transformation rule is instantiated and replaces the term *t*.

HATS has two types of transformation rules: first-order transformation rules (FOTR) and second-order transformation rules (SOTR). An FOTR matches terms and produces new terms. For example, the following rule is FOTR.

$$TR\text{-}1 = i \rightarrow j$$

An SOTR matches terms and produces FOTRs. The following rule is an SOTR that generates an FOTR for each term matching the left-hand side of the rule, namely *(i j)*. Note that the operator → is right-associative.

$$TR\text{-}2 = (i\ j) \rightarrow i \rightarrow j$$

2.2.2 Control Strategies

There are two important properties in any transformational system, *termination* and *confluence*. A transformational system is terminating if it has no infinite transformation sequences, i.e., the application of the transformation rules, *T*, to a term, *t*, will eventually stop and produce a new term called *normal form*. Confluence means that in spite of the order of applying the transformation rules *T* to the term *t*, the resultant normal form is unique [GH94][T03]. Ideally, transformation rules can be applied in any order to a source file, and the transformation will terminate and produce a unique normal form. However, this is usually not the case since some transformation rules may give rise to infinite branches and non-confluent systems may produce different normal froms. Thus, it is necessary to have strategies that control the application of the transformation rules to the source files.

In HATS, the transformation rules are applied according to four control strategies: *fix*, *once*, *transient*, or *hide*. In fix, the rules are applied in an exhaustive fashion. An SDT that represent a source file is traversed, and the transformation rules are applied to every node. This process is repeated until no more application of the transformation rules is possible. The strategy once traverses the SDT only once. The transient strategy traverses the SDT and is applied at most once, i.e., after the first application of a transformation rule using the transient strategy to a node in the SDT, the rule is reduced to a *skip* combinator that prevents it from being applied any more. The hide strategy prevents the HATS engine from knowing whether the application of a certain rule is successful or not.

Each call of fix, once, transient, or hide takes three arguments: a traversal mode, a source file, and a transformation sequence. The traversal mode determines the way that the control strategy will walk through the SDT of the source file. The purpose of the traversal is to apply the transformation rules, which are represented by the transformation sequence, to the SDT in a certain way. Two traversal modes are supported in HATS: pre-order and post-order. In the pre-order traversal mode, parents of nodes are visited first, and then the siblings are visited. In the post-order traversal mode, the siblings are visited first, and then the parent is visited.

2.2.3 Combinators

HATS supports three combinators: sequential (;), left-biased (<+), and right-biased (+>). If a sequence of transformation rules is composed using the sequential combinator, then the sequence will be executed-to-completion on a given term, *t*, from left to right. For example, if one executes the sequence $(r_1;r_2;r_3)$, on the term *t*, the rule r_1 will be matched with the term *t* and the term *t'* is produced. The second rule r_2 will be tried on t', and so on. The left-biased combinator, on the other hand, will proceed from left to right, but it will stop as soon as one of the rules in the sequence is successfully applied. The right-biased combinator is similar to the left biased combinator except that it starts executing from right to left and stops when a rule in the sequence is successfully applied.

2.3 Example

Consider the following association list representing a table.

```
table =  ((1 “Hello”)
          (2 “World”)
          (3 2)
          (4 3))
```

Each entry of `table` consists of two components: the first is a natural number, and the second is either a string 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 `table`. The goal in this example is to resolve the pointers in the second column of `table`, i.e., to replace each index with the string to which it points. To achieve our goal, we need the following sequence of FOTRs.

```
TR-1 = (x 1) → (x “Hello”)
TR-2 = (x 2) → (x “World”)
TR-3 = (x 3) → (x 2)
TR-4 = (x 4) → (x 3)
```

We will refer to the above sequence as `rule-list`. In this example, the rules in the sequence `rule-list` are composed using the sequential combinator. The application of the sequence `rule-list` to the `table` once will yield the following result.

```
once(rule-list, table) =  ((1 “Hello”)
                           (2 “World”)
                           (3 “World”)
                           (4 2))
```

The application of the *once* strategy only one time is not enough to resolve all the pointers in the `table` since applying the `rule-list` to the `table` once will leave, in the above case, one unresolved pointer. Thus, we need to use the fix strategy as follows.

```
fix(rule-list, table) =  ((1 “Hello”)
                          (2 “World”)
                          (3 “World”)
                          (4 “World”))
```

3 Motivating Example: The Sandia Secure Processor (SSP)

SSP is a project at Sandia National Laboratories and is intended to provide a general-purpose computational infrastructure suitable for use in high-consequence embedded systems. It is a simplified Java processor that consists of two components: the SSP-classloader, which is implemented in software, and SSP-runtime, which is implemented in hardware.

3.1 SSP vs. JVM

The JVM specifications provide for applications to be executed before all class files are completely loaded. This execution eagerness is preferable in applications that have class files spread over the Internet. However, in embedded systems this poses great risk. Therefore, the SSP is a closed system in the sense that all the classes that will be used in the execution must be available before the execution starts, and the class loading activities of the JVM can be performed statically. Figure 3-1 shows the relation between the JVM and the SSP.

The SSP does not implement the following features of Java: garbage collection, multiple threads, interfaces, exception handling, floating point operations, dynamic arrays, initialization of static fields, and Java libraries.

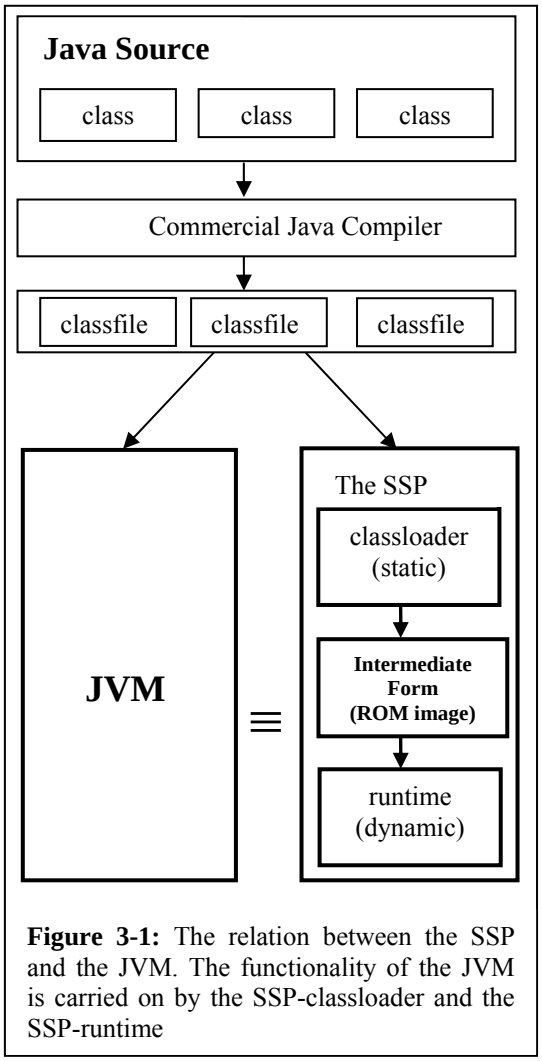

Figure 3-1: The relation between the SSP and the JVM. The functionality of the JVM is carried on by the SSP-classloader and the SSP-runtime

3.2 The SSP-classloader

The functionality of the SSP-classloader can be decomposed into a sequence of canonical forms in which a class file, C_0, is transformed to a ROM image, C_{ROM}. Each of the transformation rules that embody a canonical form manipulates the source file and produces an intermediate form that can be used by the next form. In order to apply this idea, a term language supporting suitable intermediate class file forms has to be designed; such a term language is shown in Figure 3-2.

In this section, we introduce the five canonical forms that embody the functionality of the SSP-classloader with concentration on the first form. The interested reader can consult [WR04A] [WR04B] [WR04C] for more details.

3.2.1 Canonical Form 1: Index Resolution

In a typical class file, various type of information is stored as indexes into the constant pool. Indexes into the constant pool may exist in several forms: references to field, method, this-class, and super-class. These indexes denote directly or indirectly information that is ultimately utf8 strings.

```
constant_pool                 ::= cp_info_list
cp_info_list                  ::= cp_info cp_info_list | ()
cp_info                       ::= [access] base_entry
access                        ::= [offset] index

base_entry                    ::= constant_name_and_type_info
                                | constant_fieldref_info
                                | constant_methodref_info
                                | constant_class_info
                                | constant_integer_info
                                | constant_utf8_info

constant_name_and_type_info   ::= name description
constant_fieldref_info        ::= class name_and_type
constant_methodref_info       ::= class name_and_type
constant_class_info           ::= name
constant_integer_info         ::= bytes
constant_utf8_info            ::= utf8

class                         ::= name
name                          ::= data
name_and_type                 ::= data
description                   ::= data
data                          ::= index | utf8 | name description
```

Figure 3-2: A context-free grammar that describes a term language appropriate to be used to represent the intermediate class file forms

To arrive at this form, all indirection in a constant pool of a Java class file, C_0, are resolved. The result is a new intermediate file, C_1. Consider the constant pool fragment shown in Figure 3-3 to which we refer as C_0. After resolving the indexes in this constant pool the result will be as shown in Figure 3-4 to which we refer as C_1.

Index	Entry Type	Contents
1	CONSTANT_Fieldref_info	2 3
2	CONSTANT _Class_info	4
3	CONSTANT _Name_And_type_info	5 6
4	CONSTANT _Utf8_info	B
5	CONSTANT _Utf8_info	Y
6	CONSTANT _Utf8_info	I

Figure 3-3: A constant pool description of the integer field B.x

Index	Entry Type	Contents
1	CONSTANT_Fieldref_info	B Y I
2	CONSTANT _Class_info	B
3	CONSTANT _Name_And_type_info	Y I
4	CONSTANT _Utf8_info	B
5	CONSTANT _Utf8_info	Y
6	CONSTANT _Utf8_info	I

Figure 3-4: The constant pool shown in Figure 3-3 after index resolution

3.2.2 Canonical Form 2: Static Field Address Calculation

The goal of this form is to assign each static field within the Java application a unique absolute address. The number of the static fields remains constant during runtime since they are associated with a class rather than objects.

3.2.3 Canonical Form 3: Instance Field Offset Calculation

The goal of this form is to assign a unique offset to each instance field within a class file. Instance fields are different from static fields. The former is associated with an object rather than a class. Thus, each object has its own copy of its own instance fields plus all the instance fields inherited from its super class.

3.2.4 Canonical Form 4: Method Table Construction

The goal in this form is to construct a method table for every class. The entries of each table contains the data necessary to execute the bytecodes corresponds to the implementation of a method. Each class must store the inherited or redefined method in the same relative position in their method table.

3.2.5 Canonical Form 5: Inter-class Absolute Address and Offset Address Distribution

The goal in this form is to distribute the absolute addresses and the offset addresses between the class files within a Java application. This is necessary because a single class file may contain symbolic references to fields and methods defined in other class files. Therefore, if a class file X has symbolic references to a class file Y, these symbolic references appear in the constant pool of X and must be resolved using the information originating from the class Y. In other words, absolute addresses and offset addresses must be transmitted from the class they are defined in to all classes they referenced from.

4 Verification

This section describes our approach to modeling and verifying the correctness of the TLPs of HATS that represent the five canonical forms of the SSP-classloader. A model of a simplified version of TLP_1, which represents the first canonical form, will be taken as an example. Figure 4-1 shows an abstract classfile before it is submitted to TLP_1. Figure 4-2 shows the abstract class file in Figure 4-1 after TLP_1 has been applied to it.

In our work, however, we are modeling and verifying a simplified problem, which is the table resolution problem in presented in section 2.3. Our work can be summarized as

follows: First, the recursive function `fix-strategy`, which models the behavior of the simplified version of TLP_1, is created. Second, a proof is provided to show that the function has a measure that decreases in each recursive call. Third, a proof is presented that the application of the `fix-strategy` to an input class file will preserve the semantics of the class file, i.e., the transformation rules, when applied to the class file in a certain order, will preserve correctness. To achieve this goal, a semantic function that describes the behavior of TLP_1 must be identified.

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()

Figure 4-1: An abstract class file before submission to TLP_1.

This class	1
Super Class	19
CP	(1,A)(2,A.x1)(3,x1)(4,A.x2)(5,x2)(6,A.a1)(7,A.a2)(8,a1)(9,a2)(10,B.x1) (11,B)(12,B.foo)(13,foo)(14,x3)(15,bar)(16,A.x3)(17,A.foo)(18,A.bar)(19,Obj)
Static fields	2@- 4@- 16@-
Instance fields	6:- 7:-
MT	
Methods	17() 18()

Figure 4-2: An abstract class file after being processed by TLP_1.

In our model, however, parsing will be done manually and the SDTs, which correspond to TLP_1 and the class file, are inserted as direct inputs. Therefore, the HATS parser is assumed to be correct since the verification of the correctness of the HATS parser is beyond the scope of our work.

4.1 ACL2 Specifications

The functionality of TLP_1 that represents the first canonical form of the SSP-classloader can be abstracted to the table resolution example introduced in Section 2.3. Our model is built around that example; however, in this discussion we will refer to the `table` as `classfile`. Figure 4-1 illustrates the dependency among the main functions involved in our model.

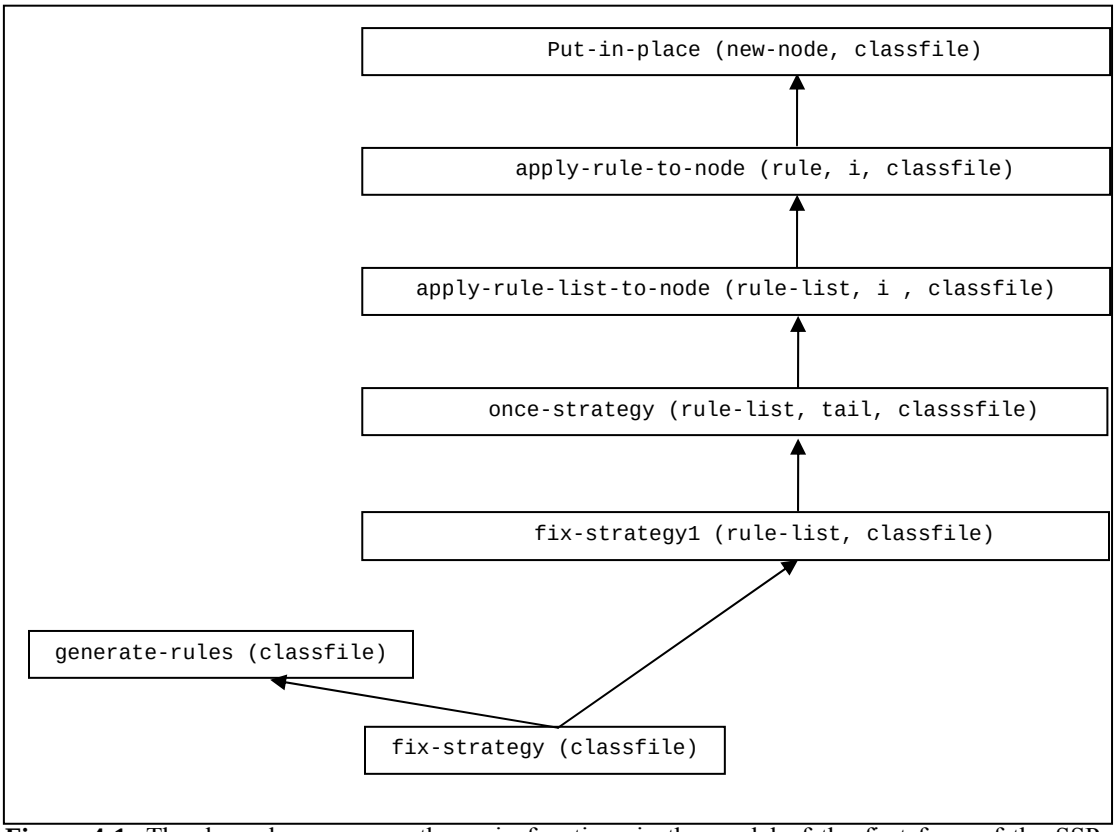

Figure 4-1: The dependency among the main functions in the model of the first form of the SSP-classloader

Our ACL2 model of TLP_1 consists of 22 functions and predicates. The description of the main functions is presented in Table 4-1. For the complete model, please see the attached script.

4.2 Verification

The verification effort consists of two parts: proving that the function `fix-strategy` terminates and that it preserves the semantics of the input `classfile`. Since the function `fix-strategy` is non-recursive, our concentration will be on the function `fix-strategy1`. To prove that the function `fix-strategy1` terminates, a measure that decreases each time there is a match between a rule in the `rule-list` and the `classfile` should be identified. To prove that the function `fix-strategy1` preserves the semantics of the `classfile`, a *semantic function*, *S*, should be identified. This function determines the equivalence of the input `classfile` and the output that results from applying `fix-strategy1` to `classfile`, i.e., the transformation rule and the respective control strategy preserve the semantics of the input classfile. Therefore, for the function fix-strategy, which models TLP1, there is a semantic function, S1. Our main conjecture for the ACL2 model of the first form is as follows.

$$\forall(C_0)\ S_1(C_0) = S_1(\texttt{fix-strategy}\ (C_0)),$$

where S_1 is the semantic function for the model of TLP_1, C_0 is the input class file, and `fix-strategy` is the model of the TLP_1 that represents the simplified first form.

Table 4-1: The description of the main function in our model

Function Name	Description
`fix-strategy`	**Input:** `classfile`
	Output: an updated copy of `classfile` in which all the pointers have been resolved.
`generate-rules`	**Input:** `classfile`
	Output: a list of FOTR rules, `rule-list`, which will be used to resolve the pointer of the `classfile`. (this function simulates how HATS generate an FOTRs on the fly from a given `classfile`.)
`fix-strategy1`	**Input:** `classfile, rule-list`
	Output: an updated copy of `classfile` in which all the pointers have been resolved.
`Once-strategy`	**Inputs:** `rule-list, tail,` and `classfile`
	Output: an updated copy of `classfile` in which one level of the pointers has been resolved.
`apply-rule-list-to-node`	**Inputs:** `rule-list, i,` and `classfile`
	Output: an updated copy of `classfile` in which one level of the pointer in the i^{th} entry has been resolved. (The updated copy is the result of applying all the possible transformation rules in `rule-list`.)
`apply-rule-to-node`	**Inputs:** `rule, i,` and `classfile`
	Output: an updated `classfile` in which one level of the pointer in the i^{th} entry has been resolved. (The updated copy is the result of applying the current transformation rules in `rule-list`.)
`put-in-place`	**Inputs:** `new-node` and `classfile`
	Output: a new `classfile` in which a modified new node, namely `new-node`, is inserted back in its correct location.

4.2.1 Proof of Termination

Admitting the recursive function `fix-strategy1` requires identifying a *measure* that decreases after each recursive call. The measure is the sum of the pointers in the second column of the input `classfile` such that if the pointer is a string, it is given the weight zero; however, if the pointer is a natural number, it is given a weight that equal to its value plus one. This measure is represented by the function `sum-addr-to-resolve` that has the following definition.

```
(defun sum-addr-to-resolve (classfile)
  (if (endp classfile)
      0
    (+ (sum-classfile-entry (car classfile))
       (sum-addr-to-resolve (cdr classfile)))))
```

The function sum-addr-to-resolve recursively accumulates the weight of the second column of each entry in the classfile using the function sum-classfile-entry, which is defined as follows.

```
(defun sum-classfile-entry (entry)
  (if (natp (cadr entry))
      (1+ (cadr entry))
    0))
```

Before proving that the function sum-addr-to-resolve decreases, we must show that it always returns an integer that is greater than or equal to zero. The lemma is stated as follows.

```
(defthm sum-addr-to-resolve-type
  (and (integerp (sum-addr-to-resolve classfile))
       (<= 0 (sum-addr-to-resolve classfile)))
  :rule-classes :type-prescription)
```

To prove that the function sum-addr-to-resolve decreases, three predicates should be defined: matches, which returns t if there is a match between a rule and a certain node in the classfile; matchp, which returns true if there is a match between any rule in the rule-list and a node; and all-matchp, which returns true if there is a match between any rule in the rule-list and any node in the classfile.

The main conjecture in the proof of termination is as follows.

```
(defthm sum-addr-once-strategy-strictly-<
  (implies
     (and (well-formed-classfilep classfile)
          (all-matchp rule-list tail classfile))
    (< (sum-addr-to-resolve
              (once-strategy rule-list tail classfile))
       (sum-addr-to-resolve classfile)))
  :rule-classes :linear)
```

In this conjecture, the predicate well-formed-classfilep returns t if every node in classfile satisfies the following constraints: the first component is a natural number and the second component is either a string or a natural number that is strictly less than the first component. This constraint is necessary to avoid loops in the classfile and make the proof of termination of fix-strategy1 easier in our simplified version (we intend to remove this restriction in our future work see section 5.) The conjecture states that: given a well-formed classfile with at least one node that matches any rule in rules-list, then the sum of the second column of the new classfile is less that the sum of the original classfile. Proving this conjecture requires proving three other similar lemmas about the functions apply-rule-list-to-node, apply-rule-to-node, and put-in-place. The three lemmas are as follows.

```
 (defthm sum-addr-apply-rule-list-to-node-strictly-<
  (implies
     (and (well-formed-classfilep classfile)
          (matchp rule-list n classfile))
     (< (sum-addr-to-resolve
          (apply-rule-list-to-node rule-list n classfile))
        (sum-addr-to-resolve classfile)))
  :rule-classes :linear)

(defthm sum-addr-apply-rule-to-node-strictly-<
  (implies
     (and (matches rule n classfile)
          (well-formed-classfilep classfile))
     (< (sum-addr-to-resolve (apply-rule-to-node rule n classfile))
        (sum-addr-to-resolve classfile)))
  :rule-classes :linear)

(defthm sum-addr-put-in-place-strictly-<
  (implies
     (and (well-formed-classfilep classfile)
          (natp (cadr (assoc n classfile)))
          (or (stringp x)
              (< x (cadr (assoc n classfile)))))
      (< (sum-addr-to-resolve
           (put-in-place (list n x) classfile))
         (sum-addr-to-resolve classfile))))
```

4.2.2 Semantic Function

To verify that the transformation rules are correctness-preserving, a semantic function is identified. In this case, the function is `get-constant`. This function will be used to prove that the transformation rules preserve the correctness of an input `classfile`. The function `get-constant` takes two inputs: a natural number `n` and a `classfile`. The chain of pointers is followed until a string is reached, and the string is returned. If no string is found, `nil` is returned. The definition of the function `get-constant` in ACL2 is as follows.

```
(defun get-constant (n classfile)
  (let ((temp (assoc n classfile)))
    (cond ((null temp) nil)
          ((stringp (cadr temp)) (cadr temp))
          ((or (not (natp n))
               (not (natp (cadr temp)))
               (<= n (cadr temp)))
           nil)
          (t (get-constant (cadr temp) classfile)))))
```

4.2.3 Main conjecture

The next step is to use `get-constant` to prove that the semantic of `classfile` is preserved after applying the function `fix-strategy`. Our main conjecture is as follows.

```
(defthm get-constant-n-fix-strategy1
  (implies (well-formed-classfilep classfile)
           (equal (get-constant n
                                (fix-strategy1 rule-list classfile))
                  (get-constant n classfile))))
```

The conjecture states that given a well-formed `classfile`, the result of following a pointer `n` in the result of applying the function `fix-strategy1` to a `classfile` will be the same as following the pointer `n` in the original `classfile`. To prove the main conjecture, we need to prove four similar conjectures about the functions: `once-strategy`, `apply-rule-list-to-node`, `apply-rule-to-node`, and `put-in-place`. These four lemmas are presented below and the interested reader can refer to the attached script for more details. For the record, to prove the termination conjecture and the semantic equivalence conjecture, 39 lemmas are needed.

```
(defthm member-position-gc-put-in-place-general
  (implies
     (and (well-formed-classfilep classfile)
          (well-formed-classfile-entryp (list position any))
          (equal (get-constant position classfile)
                 (if (stringp any)
                    any
                  (get-constant any classfile))))
     (equal (get-constant n
                  (put-in-place (list position any) classfile))
            (get-constant n classfile))))

(defthm get-constant-apply-rule-to-node
  (implies (well-formed-classfilep classfile)
           (equal (get-constant n
                       (apply-rule-to-node rule position classfile))
                  (get-constant n classfile))))

(defthm get-constant-n-apply-rule-list-to-node
  (implies
     (well-formed-classfilep classfile)
     (equal (get-constant n (apply-rule-list-to-node rule-list
                                              position classfile))
            (get-constant n classfile))))

(defthm get-constant-n-once-strategy
  (implies (well-formed-classfilep classfile)
           (equal (get-constant n
                     (once-strategy rule-list tail classfile))
                  (get-constant n classfile))))
```

5 Conclusion and Future Work

The proof of correctness of the simplified version of the first form shows the applicability of our approach to verify that the TLPs preserve the correctness of the input structure. Our formal approach to model and verify a simplified version of TLP_1 of the SSP-classloader using ACL2 establishes a framework which can be adapted in verifying other HATS implementations. However, it should be clear that the semantic functions, which are a crucial part of the verification effort, might vary depending on the applica-

tion. This suggests that significant effort must be devoted to understanding the source file, the TLP, and the output. When the semantic function is defined, then key lemmas can be identified. We believe that there is a general pattern among HATS applications. Thus, verifying one application can help in verifying others. Furthermore, we believe that this framework is applicable to other TOP examples.

Our next step is to remove the restriction on the input table so that the second component, if it is a natural number, should not be less than the first component. A problem that results from removing this restriction is that we have to avoid loops, i.e., a pointer x leads to a pointer y and the pointer y leads to the pointer x again. Next is to allow the second component of each entry of the table to be a list of two components in order to more closely match actual Java classfiles.

Acknowledgment

The authors are grateful to J Strother Moore for his significant help and support since the beginning of this work. He gave us the opportunity to work with the ACL2 group to get the necessary training and spared time to meet with us and read our early version of the introduced model. We especially thank Victor Winter at the University of Nebraska at Omaha for his comments and suggestions regarding our model of the first form of the SSP-classloader, Hanbing Liu who spared no effort to guide us in modifying a part of the model and proving some of the main theorems, Jared Davis for his many comments and suggestions. We also thank Matt Kaufmann for his suggestions to improve our model and the ACL2 group at the University of Texas at Austin.

References

[BS93] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *IEEE Software Engineering Journal*, Vol. 8, No. 4, July 1993, pp. 189–209.

[GH94] D. Gabbay and C. Hogger, et al., *Handbook of Logic in Artificial Intelligence and Logic Programming: Deduction Methodologies*, Oxford Press, Vol. 1, 1994.

[KM00a] M. Kaufmann, P. Manolios, and J S. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June 2000.

[KM00b] M. Kaufmann, and P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: Case Studies,* Kluwer Academic Publishers, June 2000.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specifications*, 2ed Ed., Addison-Wesley, Reading, Massachusetts, 1999.

[RH04] P. Regan and S. Hamilton, "NASA's Mission Reliable," Computer, *IEEE computer society*, Vol. 37, No.1, 2004, pp. 59-68.

[T03] Terese (Eds.), *Term Rewriting Systems*, Cambridge Press, 2003.

[WR04A] V. Winter, S. Roach, and F. Fraij et al., "High-Order Strategic Programming: A Road to Dependable Software," Submitted to *IEEE Transactions on Dependable and Secure Computing*, 2004.

[WR04B] V. Winter and S. Roach et al., "The SSP: An Example of High-Assurance Systems Engineering," *Eighth IEEE International Symposium on High Assurance Systems Engineering*, Tampa, Florida, March 25-26, 2004.

[WR04C] V. Winter, S. Roach, and F. Fraij et al., "High-Order Strategic Programming: A Road to Dependable Software," Submitted to *ACM Special Issue on Embedded Systems*, 2004.

[WR03] V. Winter and S. Roach, et al., "Transformation-Oriented Programming: A Development Methodology for High Assurance Software," in Marvin Zelkowitz (Ed.): *Advances in Computers: Highly Dependable Software*, Vol. 58, 2003.

[W99] V. Winter, *User Manual for HATS 1.3*, Sandia National Laboratories, 1999.