Transformation-Oriented Programming: A Development Methodology for High Assurance Software*

Victor L. Winter
University of Nebraska at Omaha
vwinter@mail.unomaha.edu

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

Greg Wickstrom
Sandia National Laboratories
glwicks@sandia.gov

Abstract

A software development paradigm known as Transformation-Oriented Programming (TOP) is introduced. In TOP, software development consists of constructing a sequence of transformations capable of systematically constructing a software implementation from a given formal specification. As such TOP falls under the category of formal methods.

The general theory and techniques upon which TOP is built is presented. The High Assurance Transformation System (HATS) is described. The use of the HATS tool to implement a portion of the functionality of a classloader needed by the Sandia Secure Processor (SSP) is described.

Contents

1 Background ................................................................. 4
  1.1 Chapter Overview .................................................... 4

2 High-Consequence Systems ............................................. 5
  2.1 Building Software is Deceptively Hard .......................... 6
  2.2 Software Development Risks ...................................... 7

*This work was 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.
3 Approaches to Developing High-Assurance Systems
3.1 Imperative Programming .............................................. 9
3.2 Object-Oriented Programming ..................................... 9
3.3 Functional Programming ........................................... 10
3.4 Aspect-Oriented Programming .................................... 10
3.5 Declarative Programming ........................................... 11
3.6 Formal “Methods” .................................................... 11
3.7 Formal Specifications ............................................... 12
3.8 Theorem proving ..................................................... 13
3.9 Model Checking ...................................................... 14
3.10 Synthesis ............................................................. 14

4 Transformation-Oriented Programming ........................ 15
4.1 Example ..................................................................... 18
4.2 Example ..................................................................... 19

5 TOP As a Program Development Method ..................... 21
5.1 Contrasting TOP with Component-based Software Development .... 22
5.2 History of TOP ........................................................ 24
5.3 Transformation Systems General Architecture ................ 25
5.4 Syntax and Semantics of Transformation Rules ................. 25
5.4.1 Example 3 .......................................................... 27

6 HATS ........................................................................... 28
6.1 Writing Transformation Rules ...................................... 29
6.1.1 Example 4 .......................................................... 29
6.2 Dynamic Transformations ............................................ 31

7 Embedded Systems ......................................................... 34
7.1 The SSP Project ....................................................... 35
7.2 The SSP-classloader ................................................ 36
7.2.1 The SSP-runtime ............................................... 37
7.3 A Transformation-based Classloader .............................. 39
7.3.1 Constant Pool Resolution ...................................... 40
7.4 Removal of Indirection .............................................. 41
7.5 Relevant Constant Pool Construction ............................ 51
7.6 Offset Indexing ......................................................... 53

8 Future Work: Verification .............................................. 55
8.1 Design Verification: Eval and BCE ................................ 56
8.2 Basis for Verifying the Partial Classloader Design ............ 60
8.3 Implementation Verification .......................................... 61
9 Summary and Conclusion 61
A A Small Java Program 62
B Java Classfile 64
C Resolved Classfile 65
1 Background

Computer-enhanced problem solving has evolved dramatically over the past 50 years. When viewed in isolation, a computer is a device, that when presented a problem expressed in a binary language, will compute an answer also expressed in a binary language. While increases in processor speed and memory have been nothing short of astounding, the fundamental model of a computer as a function mapping binary inputs (i.e., programs and input data) to binary outputs has essentially remained unchanged. However, what have changed dramatically over the past 50 years are the techniques and approaches that can be employed to produce these binary inputs – in particular, binary programs.

Initially, the construction of binary programs was undertaken entirely by humans. Conceptually simple computations had to be described in terms of long sequences of 0’s and 1’s. Such program development was highly error prone. Even the slightest slip of the finger on the keyboard (or punchcard) would cause the program to fail. Furthermore, discovering the root cause of failure in programs written in machine code could require extremely complex analysis. Because of the cryptic notation and nature of feedback provided, the class of problems that could be effectively solved in this paradigm (i.e., the programs constructed) was (of necessity) quite modest when compared to modern standards. To assist in the development of software, tools were developed. Many of these tools partition software development into segregated activities where the responsibility of accomplishing a particular task fell predominantly upon either the developer (e.g., design) or the tool (e.g., compilation). Thus a crisp boundary existed between human tasks and computer tasks.

However, a dramatic blurring of this boundary is occurring. A plethora of tools and notations have been developed to assist the developer in virtually every aspect of software development including problem conceptualization, requirements elicitation, specification, design, code generation, and verification and validation. As a result, human activities and tool activities are becoming intertwined within software development paradigms. This chapter is devoted to transformation-oriented programming (TOP), an intertwined software construction paradigm for which formal verification can be used to provide assurance of correct software behavior.

1.1 Chapter Overview

This chapter has two parts. The first half is a general introduction to various paradigms of programming with particular attention given to TOP. The second half is a more detailed description of a specific tool called HATS and an application demonstrating its use. In the following section, we discuss high consequence systems and motivate the need for improvement in software development practices, in particular with respect to providing evidence in correct behavior. Section 3 describes current approaches to software development and puts transformation-oriented programming in the context of current work.
Section 4, *Transformation-Oriented Programming*, introduces transformations and rewriting as a mechanism for describing the conversion of data from one form to another. Section 5 extends the idea of transformation-oriented programming to include the creation of programs. The fundamental idea is that given a formal specification, a correct program can be created using a transformation tool. Section 6 describes a particular transformation tool called *HATS*, the High-Assurance Transformation Program.

In Section 7 we describe a specific application of transformation-oriented programming, the Sandia Secure Processor (SSP). This processor is intended to be an embedded processor interpreting a subset of the Java bytecode language for high-assurance applications. Transformations that implement the classloader for this processor are presented. Section 8 outlines work that must be done in order to validate the code generated by the transformations.

## 2 High-Consequence Systems

Applications in which there is a high cost associated with failure are called *high-consequence systems*. In addition, if the cost of failure is measured in terms of human life we call the system a *safety-critical system*. In order for a high-consequence system to be certified for use, it typically must be demonstrated that the likelihood of a failure occurring during any given operational hour is on the order of 1-in-$10^9$. Such a system can be expected to fail only once every 114,155 years.

*High-assurance software* is software for which there is convincing evidence that the software possesses certain explicitly stated attributes (e.g., reliability). The attribute that one is generally most interested in here is *correctness*, which is the notion that the behavior of the software is in compliance with its intent (not just its specification). Because of the dominance of correctness as an attribute, the phrases *high assurance* and *high assurance of correctness* are often used synonymously. A *high-assurance system* is a system for which convincing evidence has been provided showing that its behavior will not result in a failure.

In spite of the dangers, software is being used at an accelerating rate to control systems that have the potential to affect the safety and well being of large numbers of people. For example, software plays a critical role in numerous systems such as antilock brakes, flight control systems, weapons systems, nuclear power plants, electrical power grids, and a wide variety of medical systems such as those for embedded cardiac support. As a result, the dependence of modern society on safety-critical software has become deeply rooted.

As society’s appetite for *high-consequence* software increases, the importance of developing *high-assurance* software systems becomes essential. A key component of any construction technique claiming to produce high-assurance software is the ability to provide sufficiently convincing evidence, prior to fielding the system, that the software will function in its intended manner. In other words, when developing a high-consequence software system, it is not sufficient to merely build a dependable system. One must
also be able to convince the stakeholders (e.g., society) that this system will function correctly if and when it is put into operation.

In this chapter, we assume that high-assurance software will be developed in two phases: a *formalization phase* and an *implementation phase*. Figure 1 gives an overview of these two phases. The goal of the formalization phase is to understand an informally stated problem and express it in a formal framework. The artifact produced by the formalization phase is a formal specification. The implementation phase uses this formal specification as the basis for constructing an *implementation*. Here we define an implementation as representation of software that is either in an executable form or in a form that can be automatically translated (e.g., via a compiler) into an executable form. In the implementation phase, we assume that the formal specification is correct. This implies that the implementation phase will be considered successful if the software system developed satisfies the formal specification.

![Figure 1: High-Assurance Software Development Phases](image)

Though they are extremely important, the tools and techniques that can be brought to bear in the formalization phase of software development are beyond the scope of this chapter. Instead, our focus is on the implementation phase as it relates to high-assurance system development. When given a formal specification, our goal is to construct an implementation in which convincing evidence can be provided that the implementation satisfies the formal specification.

### 2.1 Building Software is Deceptively Hard

*Software is malleable.* This is both its strength and weakness and has often resulted in unrealistic expectations regarding (1) what software can and cannot do, (2) how easily something can be done in software, and (3) the level of effort required to effectively modify a software system. Unlike other engineered products (e.g. a bridge or an airplane),
modification of software does not require the replacement of tangible parts [18]. The malleable nature of software has frequently resulted in system redesigns where many mechanical controls are re-implemented in software. With mechanical controls, safety interlocks are common. If a mechanism fails, the interlock ensures the system remains in a safe state. With software systems, these controls may fail, and the software-based interlocks may also fail, sometimes for the same reasons. An example of this is the Ariane 5 rocket failure [8]. Another failure of this type occurred in the mid 1980s, the computer-controlled Therac-25 radiation therapy machine delivered overdoses resulting in serious injury or death to at least six people. The Therac-25 reused software from the Therac-20, but the Therac-20 had independent hardware protective circuits that prevented an overdose. The Therac-25 depended solely on software [29]. In a well designed system such as commercial aircraft, a single point of failure does not result in a system failure. However, in software, a failure can frequently be traced to a single point. When considered in the context of a physical system, this can contribute to system failure. In this chapter, we restrict our attention simply to software failures.

Software systems can easily become overly complex. “Computers often allow more interactive, tightly coupled, and error-prone designs to be built, and thus may encourage the introduction of unnecessary and dangerous complexity” [29]. A significant source of complexity arises from coupling of processes, which Chiles[10] and Dorner[13] independently suggest is difficult or impossible for humans to fully grasp. Analysis of major accidents such as nuclear power plant accidents and airline crashes invariably shows that these accidents are not caused by single component failures, but by highly complex, cascading sequences of events.

Software is buggy. The increased complexity in both product and process give rise to many new hazards. Oftentimes testing is used in hopes that hazards and errors will be discovered. Unfortunately, as the complexity of systems increase, a point of diminishing returns is reached. “Microsoft released Windows XP on Oct. 25, 2001. That same day, in what may be a record, the company posted 18 megabytes of patches on its Web site: bug fixes, compatibility updates, and enhancements”[32]. Microsoft is not alone. In [45] it was reported to the National Institute of Standards and Technology that 30%-90% of labor expended to produce a working program is testing. Furthermore, the findings of the report indicated that even when using software engineering best practices, when counting all errors found during design, implementation, unit testing, integration testing, and maintenance of a software system, 5% of the errors are only discovered after the product is released. In 1999, the estimated cost due to software error in the aerospace industry alone was $6 billion dollars.

2.2 Software Development Risks

In a well-planned software project, the tools used, the techniques applied, and the resources allocated to various development activities should be appropriate for the actual risks faced in the project. In order to accomplish this, one must have a clear understanding of the nature and scope of the risks encountered in software development as well
as what tools and techniques can be best applied to mitigate these risks. Overly optimistic misconceptions of software is a recipe for disaster, especially for high-consequence systems.

Abstractly speaking, risk is a combination of the likelihood of an undesirable event and the severity of its consequences. What constitutes risk with respect to a particular software product depends largely on the nature of the product as well as a given perspective (e.g., economic, environmental, etc.). For example, when developing commercial software in a highly competitive marketplace, \textit{time to market} may be the most important factor in the risk equation from the standpoint of sales and profitability. In contrast, when developing software for niche markets, \textit{development cost} may be the dominating factor. And when developing software for high-consequence systems, the primary factor is the ability to achieve high-assurance; that is, to convincingly demonstrate that the system will function as intended. A high-consequence software system for which such high-assurance cannot be provided is useless at best (because it will not be fielded) and extremely dangerous at worst (it is fielded and a failure is encountered).

So how does one address the risks associated with high-consequence software development? To date, some of the most promising tools and techniques for constructing high-assurance software are rooted in a form of mathematics that is known as \textit{formal methods} \cite{46}\cite{7}\cite{24}. The basic idea when using a formal method is to construct a particular artifact (e.g., a specification, or an implementation) in such a way that mathematical reasoning (i.e., calculation) can be used to answer certain questions – the most ambitious of which would be, “Is this program correct?”.

\section{Approaches to Developing High-Assurance Systems}

There are many approaches to the development of software systems, and there are a large number of tools and notations available to assist software developers. Finding the right approach and using the right tools to develop a particular software system is key to achieving our software development goal. This section takes a brief look at some of these approaches. We use the term \textit{paradigm} to mean a general approach to problem solving facilitated by a particular notation. In contrast, a \textit{method} is defined here as a notation accompanied by a process. A process suggests a sequence of steps that a developer takes in order to produce some artifact such as a design or a test case. The notations and processes used influence how developers think about problems and software solutions. Frequently, different paradigms and methods can be combined, each being used to solve different parts of a larger problem.

Regardless of the paradigms or methods used to develop a software system, in order to successfully construct a software system, the following three steps must be accomplished: (1) the requirements of the system must be established; (2) the software must be designed; and (3) the design must be translated into machine executable code. This translation is typically accomplished in part by humans writing high-level language code, and in part by computers executing compilers, translators, interpreters, and as-
semblers. It is not necessary that the three steps be completed in a particular order, nor is it necessary for each step to have an artifact associated with it. A programmer who does not write a design on paper still has a design, even if its only manifestation is in the programmer’s head. These steps may also be accomplished in strict sequence with no overlap as they are, for example, in the waterfall development process. Or they may be completed incrementally as they are in the spiral and evolutionary development processes. For high-assurance systems, there is the additional task of providing evidence that the software is correct.

3.1 Imperative Programming

It is widely accepted that the first imperative language was Fortran, and many other imperative languages followed, including Algol and Cobol [47]. In imperative programming, program variables hold the program state, and the primary unit of computation responsible for incremental change is the assignment statement. Control is dictated by various constructs such as the sequential composition operator as well as conditional statements, loops, statements, jumps, and procedure calls. This provides a way in which solutions can be incrementally constructed. Block structuring provides a way of grouping code fragments such as statement sequences, and even other blocks into a unit or module. Such groupings can facilitate the understanding of the structures within a program at higher levels of abstraction than is possible when simply viewing a program as a composition of primitive statements. The introduction of block structuring has had a major impact on the ability of programmers to construct larger more complex software systems because it enables them to envision a software system in terms of an architecture consisting of blocks rather than statements. The primary tool support to programmers using the common imperative languages is editors, compilers, and debuggers.\(^1\) The errors detected in this framework include syntax and type errors.

3.2 Object-Oriented Programming

As the problems to which software systems are being applied become more complex, software developers realize that more effort must be expended establishing the requirements and designing a solution. There is also the need to demonstrate that the software is adequate either through testing or through some other means. The management of complexity has become the primary consideration.

One approach to managing this complexity is to decompose software and encapsulate functionality into discrete units. Objected-oriented programming is the current dominant paradigm. It is an extension of imperative programming that encourages system decomposition in terms of objects and classes. The class is essentially an extension of the block structure in which a rich mechanism is provided for defining the internal state of an object as well as controlling the visibility of methods, variables, and types.

\(^1\)There are tools available that generate imperative language code such as code generators and report writers. These are considered in later sections.
Inheritance is a mechanism provided for defining new classes based on other classes. Inheritance facilitates the creation of class hierarchies in which classes leverage off of the similarities between each other, thereby maximizing code reuse and understanding. Most object-oriented languages can trace their roots to Smalltalk. Object-oriented programming tends to require a greater emphasis on design and factoring than imperative programming does. While it is possible to implement an object-oriented design in an imperative language, some languages such as Eiffel, Java, and C++ more easily support inheritance, polymorphism, and encapsulation. Modern object-oriented software development environments facilitate the creation and modification of design documents such as diagrams in the Unified Modeling Language (UML). Some of these environments assist the user in the creation of high-level language code from design specifications.

3.3 Functional Programming

A alternative approach to problem decomposition is given in the functional programming paradigm. Here, programs are decomposed into functions, each of which may be arbitrarily complex. Once a function is written, it may be used without concern for the internal details. Functional programming paradigms typically provide a mathematically clean semantic framework (e.g., call-by-value parameter passing, type completeness, referential transparency) that encourages equational reasoning. Computation is accomplished by function evaluation. A purely functional language does not require variables or assignment statements. Iteration is accomplished through the use of recursion. The origins of functional programming are rooted in the Lambda Calculus. Lisp was the first functional language, and Common Lisp continues to be widely used in the field of artificial intelligence. Most modern functional languages provide a rich set of abstraction mechanisms. For example, ML offers structures, which are akin to classes as well as functors, which can be thought of as generalizations of structures.

3.4 Aspect-Oriented Programming

Not every problem can be cleanly decomposed into cohesive pieces. The motivation behind aspect-oriented programming [14][28] (AOP) is separation of crosscutting concerns. Concerns can range from high level, such as security, to low level, such as caching and buffering. An aspect is the embodiment of a concern. Other examples of aspects are: correctness/understandability, spatial efficiency, temporal efficiency, and error recovery. In standard object-oriented programming, it is difficult to separate these concerns into classes or objects. The source code that embodies an aspect is spread throughout many objects. AOP advocates the construction of systems by first specifying the various concerns and their relationships, then relying on the environment to automatically weave the code supporting the aspect with other code, forming a complex system that satisfies all aspects simultaneously. AOP extends object-oriented and imperative programming using all techniques available to achieve their goal of separation of concerns.
3.5 Declarative Programming

Declarative programming (or logic programming) provides an altogether different paradigm for solving problems [51]. In a declarative environment, properties of a system are stated in terms of logical formulas. These formulas are typically presented in predicate logic. Prolog, the most widely used declarative language, uses Horn-clauses, a subset of predicate logic. Programs do not state how a solution is to be computed. Instead, a problem (i.e., computation) is stated in terms of a query. When presented with a query, a declarative programming system uses a complex search algorithm (transparent to the user) based on resolution and backtracking in order to find an answer to the query. Declarative environments provide a powerful (and complex) computational framework; however, the implementations of these environments tend to be complex, and the search required by the implementation tends to require semantic information (e.g., the cut) to direct search by pruning the search tree. Declarative programmers still must develop designs and algorithms, but their designs and solutions are much different than imperative and object-oriented solutions are.

3.6 Formal “Methods”

As demonstrated by logic programming, formal notations may be used to describe not only the programs, but also the designs and requirements of the software system. We believe that in situations where the cost of failure is the overriding concern, as it is in high-consequence and safety-critical applications, formal approaches to software development are not just viable, but necessary. We use the term formal method to mean a language with formal syntax and semantics and a set of rules for inferring useful information from sentences written in the language [23]. Typically, we start with a set of requirements and then prove properties about the specification, derive an implementation, or prove that a given implementation satisfies the requirements.

The use of formal methods is not the norm in industry, even for high-consequence software. As Parnas explains,

“When new methods do not catch on, there are two obvious possible explanations. Either the methods are not yet good enough, or practitioners are too conservative, unwilling to learn, and resistant to change. In most cases, there is truth in both explanations. The best known formal methods clearly work, but it is equally clear that they require a lot of tedious writing of expressions that are difficult to read” [42].

It is true that the use of formal methods is expensive, in part because it requires time-consuming formulation of problems in a mathematically rigorous and precise manner. However, after the initial cost of formulation, the use of formal methods may actually drive down the cost of systems. As further evidence of their usefulness, note that in every case where formal methods were used under a NASA Formal Methods program [9], previously unknown errors were discovered. Examples of costly errors in mission
software include the Mars Rover priority inversion and deadlock in the Deep Space 1 Remote Executive [27][21][44][22]. In spite of extensive testing and reviews using the best available methods, these errors persisted. These errors were discovered (or at least detected) using formal methods. In other words, even the best software practices deliver software with errors that can be detected using formal methods.

Formal methods work well for simple, textbook examples used in the classroom. Until recently, they have not competed well against traditional approaches for a class or size of programs of practical use, e.g., the kinds of programs that developers have succeeded at writing over the past fifty years. Current programs tend to be decomposable, with each unit significantly decoupled from other units, and they can be tested (though not completely or perfectly). However, we believe that software systems in the near future will be larger, more complex, not easily decomposed, and the components of this software will be tightly coupled with complex interactions. They will have many tedious details and require vast inputs and high speed. Some will control safety-critical systems. The system developer of the mid 21st century will require software tools to support program development, and the program support will be in the form of formal methods.

It is clear that formal methods hold great promise in the cost-effective construction of reliable software. It is equally clear that in order for formal methods to gain acceptance in the software development community, tools must continue to be developed to assist practitioners. This remainder of this section discusses some of the formal approaches that have been used successfully in industry.

3.7 Formal Specifications

A formal specification is the statement of a collection of properties that a system should satisfy, given in a formal language at some level of abstraction. The key difference between a specification and a program is that a specification must offer some abstraction of the problem. The examples of formal specifications have shifted over the past half-century. In the late 1950s, Fortran was considered a specification language. It dictated at some level of abstraction what a program was to do. The drive throughout computing history is to increase the power of translators that create executable code from specifications.

Current specification languages are much more sophisticated than Fortran. There are many such languages available, including report generator languages (fourth generation languages), model languages based on set theory and predicate logic such as Z and VDM, and algebraic specification languages such as CLEAR. Introductions to formal specifications are available [41][26][16].

A major benefit of formal specification is that many errors are caught simply by the act of specifying a system formally. For example, Mukherjee and Wichmann [40] describe the certification of a computerized voting system for the Church of England where the act of specifying the algorithm in VDM exposed several ambiguities in the English language description of the algorithm.
3.8 Theorem proving

An automated theorem prover is a software system that takes a set of formulas as input and applies inference rules to derive new formulas that are logical consequences of the original formulas. Typical inference rules include modus ponens, resolution, and induction. Theorem provers are usually associated with a given logic, such as propositional, first-order, higher-order, or Horn-clause logic. They may be fully automated, or they may rely on interaction with humans to guide the search for a proof.

In theorem proving, we care about the form of the arguments. For example, given the statement “All men are mortal,” and the statement “Socrates is a man,” we can derive the statement “Socrates is mortal.” As another example, given the two statements “All dogs are loyal” and “Uri is a dog,” we can derive “Uri is loyal.” Note that these two arguments have the same form. A theorem prover treats them in the same way.

In general, we start with a formula that is a formal specification $S$ of program behavior and a set of formulas that describe the context $C$, including assumptions about the program and its environment. Then we try to demonstrate that for some formal model of the implementation $I$, $C \rightarrow (I \rightarrow S)$. If we succeed in proving this, then we can say that $I$ is correct with respect to $S$ and $C$.

As with testing, failed proofs can serve diagnostic purposes. “Why can’t I prove it” is sometimes answered, “because I need more information” or perhaps “because it is wrong.” Either of these answers can lead us to discover errors before the error manifests itself at runtime.

A recent example of the use of theorem provers in software development is the use of the PVS [48] theorem proving system in the verification of the AAMP5 processor [50]. A portion of the instruction set and register-level transfers were described formally, and PVS was used to demonstrate that the microcode correctly implemented the specified behavior of the instruction set. During this exercise, errors were discovered both during the specification of the system and during the proof process.

The Nqthm theorem prover [39] was used to verify the implementation of an assembly-level programming language called Piton, which supports constructs such as recursive subroutine call and return, stack-based parameter passing, and arrays. Piton is implemented via a mathematical function called a downloader that accepts a Piton program as input and produces a binary image as its output. This binary image can be run on the FM9001 processor. The downloader function can be decomposed into a compiler, assembler, and linker. The theorem that is proven is essentially the following: Let $S_n$ denote the state (e.g., the answer) that is produced from running a well-formed Piton program $p_0$ for $n$ steps. Let $S_n'$ denote the state produced by downloading $p_0$ and running the resulting binary image on the FM9001 for $k$ steps. Then $S_n'$ is equivalent to $S_n$. Here, link tables are used to properly interpret $S_n'$ enabling the extraction of the answer.
3.9 Model Checking

Model checking [25][38] is a technique used to test properties of programs by verifying the property using an exhaustive finite-state search of a model of a program. Because the state space of complex programs is too large for exhaustive search, a common approach is to abstract the program to a model and test properties of the model. If it is possible for the model to enter a state where a specified property does not hold, the model checker detects the error and provides a transition sequence leading to that state. Model checking is most often used to discover errors in concurrent systems such as deadlock and data race. Testing and debugging concurrent systems are notoriously difficult, and some recent NASA projects have suffered from defects in concurrent systems [27][21][44][22].

A standard technique used to discover errors in software systems via model checking follows the sequence given below.

- The developer determines desirable properties for a program (such as the avoidance of deadlock) and specifies these in the formal language required by the model checker.
- The developer manually abstracts the program source code to a model, attempting to be faithful to the relevant characteristics of the program.
- This model is encoded in the language of a model checker.
- The model checker is run. The model checker tests the properties in each state.
- If an error is discovered, a trace explaining an execution path reaching the error is reported by the model checker. The developer then attempts to map the error back to the program and verify that a corresponding error is present in the actual program. Either the model is corrected or the program is corrected, and the process is repeated.

Model checking is based on temporal logic. Temporal logic is a logic of state sequences. Thus, in temporal logic, formulas are not statically true: the truth-value may change depending on the state. Model checkers have been used to identify errors in software and protocols for concurrent, distributed, and reactive systems. Some of the most popular model checking systems available today include SPIN, SMV, and UPPAL. An introduction to model checking and these systems can be found in Berard, et al [4].

3.10 Synthesis

The goal of program synthesis is to automate the process of transforming a formal specification to code that is executable on a computer. Program synthesis is a mechanism for elevating the task of programming from code generation to specification generation [15]. An incomplete taxonomy of synthesis mechanisms includes the following:
• **Inductive Synthesis:** In inductive synthesis, a program is obtained from a generalization of partial specifications. Given a set of both positive and negative examples, a program is generated that covers at least all the examples. Inductive synthesis usually requires a great deal of interaction with the programmer [17].

• **Deductive Synthesis:** Deductive synthesis systems constructively prove a conjecture based on the specification. The three steps in the synthesis are to construct a formula, prove it, then extract instances of values bound to existential variables as the program. The resulting program is a logical consequence of the specification and the background theory. Three fundamental methods of deductive synthesis are:

  – **Transformational Synthesis.** Transformation rules are applied to program specifications iteratively until an executable program is generated. This work is an outgrowth of optimization techniques. REFINE is one example of a transformational synthesis system [43].

  – **Schema Guided Synthesis.** Programs are designed by successive instantiation of templates. KIDS [49] and SPECWARE[37] are examples of this type of system.

  – **Proofs As Programs.** Proofs are the traditional approach to deductive synthesis [19][20][33][34]. A program is extracted from the proof. These systems take a specification of the form \( \forall x \exists y : P(x) \rightarrow R(x, y) \) and prove a theorem of the form \( \forall x : P(x) \rightarrow R(x, f(x)) \). Here, \( P \) is some set of preconditions on the input variables \( x \); \( y \) is a set of outputs; \( R \) is a set of post conditions constraining the outputs in terms of the inputs. The theorem prover constructs the term \( f(x) \), that computes the outputs. Amphion [31] is an example of this type of program.

4 **Transformation-Oriented Programming**

Transformation-oriented Programming (TOP) is a software development paradigm that encourages viewing and manipulating a program in “terms of the whole” rather than in “terms of its parts.” As such, this approach is well suited for an implementation phase that begins with a formal specification. This is one of the attributes that makes TOP a candidate for high-assurance software development.

---

**SIDEBAR**

The concept of understanding change (e.g., development) with respect to the whole is becoming widely used for commercial software development. For example, eXtreme Programming [3] advocates that software changes be
undertaken in such a manner that a functioning product is produced each
day. A similar approach is adopted by Microsoft. “A common practice
at Microsoft and some other shrink-wrap software companies is the ‘daily
build and smoke test’ process[36]. Every file is compiled, linked, and com-
bined into an executable program every day, and the program is then put
through a ‘smoke test’, a relatively simple check to see whether the product
‘smokes’ when it runs...By the time it was released, Microsoft Windows NT
3.0 consisted of 5.6 million lines of code spread across 40,000 source files.
A complete build took as many as 19 hours on several machines, but the NT
development team still managed to build every day[58]. Far from being a
misfire, the NT team attributed much of its success on that huge project
to their daily builds”.

In TOP, the basic unit effecting change is the transformational step. A term is
the data changed by a transformational step. In this context, it is helpful to think of
a term as a structured representation of data such as a parse tree corresponding to a
program string. The purpose of the term structure is to provide information (e.g., type
information or contextual information) about data. This information typically provides
the basis for defining change. How terms can be constructed, what information their
structure can contain, and support for detecting inappropriate contexts for terms are
areas currently being researched[11][12][53].

A transformation rule is a function from terms to terms. The syntax and semantics
of transformation rules are discussed in more detail in Section 5.4. In the current
section we develop some basic terminology. The application of a transformation rule T
to a term P is generally written T(P) and results in one of two possible outcomes: (1)
the application produces a transformational step, in which case P ≠ T(P), or (2) the
application fails to produce a transformational step², in which case P = T(P).

Let P₁ denote a term and T denote a transformation rule. If P₁ is constructed
from other terms, it will be possible to apply T to more than one point in P₁. We
choose any subterm x in P₁ and apply T. This results in a transformational step that
produces a transformed term P₂. It is again possible to apply T to each point in P₂.
In general for a given a set of transformation rules R, an input term P₁, a graph
can be constructed reflecting all possible transformational steps. This graph describes
a rewrite relation. In the graph, nodes denote unique terms, and directed edges denote
transformational steps. A directed edge exists between P₁ and Pⱼ if and only if there
exists a transformation rule, in R, that when applied to a particular subterm in P₁
yields Pⱼ and P₁ ≠ Pⱼ. It is worth noting that paths in this graph may be infinite and
either non-cyclic or cyclic. In Figure 2, the initial term is labeled P₁. The graph shows

²The ρ-calculus [11] is based on a different application semantics. Specifically, if an application of
a transformation to a term is inappropriate then the empty term (actually an empty set) is returned.
that if the transformation rule $R_1 \in \mathcal{R}$ is applied to $P_1$, the result is $P_2$. We label the node $P_2^{R_1}$ to indicate that $P_2$ was derived from $P_1$ by application of rule $R_1$. If instead transformation rule $R_2 \in \mathcal{R}$ is applied to $P_1$, the result is $P_2^{R_2}$. Note that in Figure 2, $P_3^{R_2 R_1}$ could also be labeled $P_3^{R_1 R_2}$.

![Figure 2: A Rewrite Relation](image)

We define a transformation sequence as a path in a rewrite relation, i.e., the sequential composition of one or more transformational steps. In this path, nodes other than the initial and final nodes are referred to as intermediate forms. In practice, a transformation sequence is realized through the controlled application of a sequence of transformation rules to some initial term.

In TOP, software development consists of manipulating terms via transformation sequences. We want to develop a set of transformation rules $\mathcal{R}$, and define a strategy for the application of rules in $\mathcal{R}$ such that the endpoint of the transformation sequence defined by the strategy is our desired term. Strategies define how often, when, and where transformation rules are to be applied. Metaphorically, they play the role of a navigator within the rewrite relation. A strategy makes precise what it means to apply a transformation rule to a term in the case where the rule is applicable in various places.

Below is an operational example of a strategy that exhaustively applies a transformation $T$ to a term $P_1$. Mathematically speaking, we say this strategy computes a fixed-point of $T$.

**Repeat**

Let $P_1$ denote the current form of the term.

Scan $P_1$ from top to bottom and apply $T$ to every point in $P_1$.

Let $P_2$ denote the result of step 2.

**Until** $P_1 = P_2$

The next sections give two concrete examples of transformation-oriented approaches to problem solving. In the examples, the intent is not to focus on notation and technical detail, but rather on the spirit of transformation-oriented problem solving.
4.1 Example

Consider the problem of translating an internally stored tree structure into a string that can be written to a file. When a tree is represented as a string, parentheses provide a way of encoding the tree’s structure. For example, a parenthesized expression having the form

\[(\text{root}\ subtree_1\ subtree_2\ldots subtree_n)\]

can be used to denote a tree having \(n\) children whose root node has the label \text{root} and whose immediate children are described by the expressions \(subtree_1\ subtree_2\ldots subtree_n\) respectively. Consider a tree consisting of five nodes, \(A, B, C, D,\) and \(E\). Node \(A\) is the root, \(B\) and \(C\) are the immediate children of \(A\), and \(D\) and \(E\) are immediate children of \(C\). The structure of the tree as well as its term representation are shown in Figure 3.

\[A\]
\[\ \ \ \ B \quad C\]
\[\ \ \ \ D \quad E\]

\[(A(B)(C(D)(E)))\]

Figure 3: Tree and term structure

For larger trees it is difficult to write the corresponding parenthesized expression if one proceeds in a strictly left-to-right fashion. (Consider a tree with 30 nodes, for example.) This difficulty arises because the relationship between a partial solution and the whole solution is not immediately obvious at any given time. (See Table 1.) In contrast, if we approach the construction of the expression from the perspective of the whole, the complexity of the problem vanishes. We proceed in a top-down manner producing a sequence of increasingly refined expressions. We initially denote the expression for the entire tree by the variable \(X_1\). We then examine the root and children of \(X_1\) and use the information obtained from this analysis to construct our first intermediate form. Suppose that the root node of \(X_1\) has the label \(A\) and that \(A\) has two children. The expression \((A\ X_{1,1}\ X_{1,2})\) describes this refinement. Here \(X_{1,1}\) and \(X_{1,2}\) are variables denoting the expressions corresponding to the children of \(A\). Parentheses are always
introduced in a balanced fashion. Furthermore, the relationship between \((A \times_1 \times_2)\) and the initial tree is straightforward. We can now refine any of the variables in our expression in the manner that we used to refine \(X_1\), producing an even more complex expression that still has balanced parenthesis. This refinement process continues until our expression is free from any variables, at which point we have our result. Table 1 below compares the left-to-right approach and the refinement-based approach for our example.

<table>
<thead>
<tr>
<th>Left-to-Right Approach</th>
<th>Refinement-based Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>((A))</td>
<td>((A \times_{1,1} \times_{1,2}))</td>
</tr>
<tr>
<td>((A)(B))</td>
<td>((A \times_{1,2}))</td>
</tr>
<tr>
<td>((A)(B)(C))</td>
<td>((A \times_{1,1,1} \times_{1,2,1}))</td>
</tr>
<tr>
<td>((A)(B)(C)(D))</td>
<td>((A \times_{1,2,1} \times_{1,2,2}))</td>
</tr>
<tr>
<td>((A)(B)(C)(D)(E))</td>
<td>((A \times_{1,2,2}))</td>
</tr>
<tr>
<td>((A)(B)(C)(D)(E))</td>
<td>((A \times_{1,2,2}))</td>
</tr>
</tbody>
</table>

4.2 Example

In this example, we consider compiling simple programs consisting of a single assignment statement into a sequence of RISC assembly instructions. This machine has a large number of registers, and the instruction set includes instructions for loading from memory to registers, storing from registers to memory, and adding the values in registers. The instruction set is described below.

- LD R1 Addr: Load register R1 with the value in memory address \(\text{Addr}\)
- ST Addr R1: Store the value in register R1 into memory address \(\text{Addr}\)
- ADD R1 R2 R3: Store the sum of the values in registers R2 and R3 into register R1

The following partial grammar describes a small language where a \emph{program} can be a high-level program or a low-level program. A high-level program consists of a single assignment of an expression to an address. A low-level program is a sequence of assembly instructions. Initially, expressions in high-level programs are either the address of a value or a term consisting of the sum of two other expressions. However, \emph{intermediate forms} of expressions are also possible – for example, an expression may be a tuple consisting of a register followed by a list of one or more assembly instructions. The addition tuples to our language reflects a design decision in which compilation is realized by passing expressions in high-level programs through intermediate forms containing tuples.
EXP ::= Addr | SUM EXP EXP | [ register, assembly_seq ]

PROG ::= ASSIGN Addr EXP | assembly_seq

Our goal is to create a transformation sequence capable of rewriting terms corresponding to high-level programs into terms representing assembly instruction sequences. In order to make our discussion more concrete, let us consider the following program as input to our transformation sequence.

initial program: ASSIGN 100 SUM SUM 101 102 103

Our plan is to transform this program into assembly instructions in the following fashion: First, we rewrite addresses to register load instructions. Then we rewrite each SUM to an appropriate add instruction. This step utilizes the [register, machine_instr_list] expression. The idea is to store the results of the evaluation of any expression into a register and explicitly keep track of that register by making it the first element of a tuple. For example, in the term [r1, a_seq], the value in register r1 represents the sum computed by a_seq. And finally, when the expression has been fully processed, we rewrite the ASSIGN term to a store instruction.

In formalizing these ideas, we make the following assumptions and notational simplifications:

1. We assume that at transformation time, the function reg() will provide the name of an unused register. (This allows us to avoid the details of register allocation in this presentation.) Further, we assume there are enough available registers to hold the needed intermediate values.

2. We assume that we have a concatenation function, +, allowing us to add statements to a statement list.

The following rewrites convert a statement in the abstract language into the machine instructions needed to execute the statement.

Rule 1: Addr → [z, “LD z Addr”] where \( z = \text{reg}() \)

Rule 2: SUM [reg1, stmtlist1] [reg2, stmtlist2] → [z, stmtlist1 + stmtlist2 + “ADD z reg1 reg2”] where \( z = \text{reg}() \)

Rule 3: ASSIGN Addr [reg stmtlist] → [stmtlist + “ST Addr reg”]
Transformation Rule 1 states that if we have an expression that is an address \textit{Addr}, we can replace this with a tuple consisting of a register followed by a single machine instruction. Note that the sum computed by the machine instruction list is stored in the register \( z \). Transformation Rule 2 states that if we have an expression consisting of a \textit{SUM} followed by two tuples, each consisting of a register and a list of statements, we replace the entire term with a tuple having a register and a list of statements. This list of statements returned is the concatenation of the two statement lists and an \textit{ADD} instruction. Again, the newly introduced register \( z \) holds the sum computed by the machine instruction list. Transformation Rule 3 states that if we have a program consisting of the terminal symbol \textit{ASSIGN} followed by an address followed by a tuple, we can replace it with a list of statements.

Below is a trace highlighting of some of the intermediate forms produced when the above transformation rules are applied using an inside-out strategy to our initial program.

1. \textbf{ASSIGN 100 SUM SUM 101 102 103}
2. \textbf{ASSIGN 100 SUM SUM [R3, LD R3 101][R2, LD R2 102][R1, LD R1 103]}
3. \textbf{ASSIGN 100 SUM [R4, LD R3 101 + LD R2 102 + ADD R4 R3 R2][R1, LD R1 103]}
4. \textbf{ASSIGN 100 [R5, LD R3 101 + LD R2 102 + ADD R4 R3 R2 + LD R1 103 + ADD R5 R4 R1][R1, LD R1 103 + ST 100 R5]}

Form 1 is the initial string. In a more standard notation, this could have the form \( d := (a + b) + c \). The terms 100, 101, 102, and 103 are addresses of the variables \( a, b, c, \) and \( d \). Form 2 is the result of applying Transformation Rule 1 to the addresses 101, 102, and 103. Forms 3 and 4 result from the application of Transformation Rule 2. Form 5 is the result of applying the last transformation. Boyle et al [5] describes another approach to this type of problem.

5 \textbf{TOP As a Program Development Method}

In Example 2 above, we demonstrated the refinement of a program from a simple high-level language into low-level assembly code. This type of approach can generalized and used to solve the problem of transforming high-level specifications of program behavior into lower level, executable programs. When discussing the transformation of specifications, programs, and the like, we say \textit{program} when referring to any term along the transformation sequence (e.g., intermediate forms as well as initial and final endpoints).
Correctness is typically an invariant property spanning the entire transformation sequence. Informally we say that an intermediate form of a program is correct if it has the same semantics as the initial program. The correctness property is the dominant influence in TOP. It requires transformational changes to be understood with respect to the entire program. It also encourages encapsulation of changes as well as separation of concerns.

A typical TOP development cycle begins with a (correct) formal specification, which we will generically call a program. Then a set of transformation rules are constructed, and the application of these rules is guided by a strategy. In this context, we consider a program to be a specification if it satisfies any of the following properties:

1. It describes an algorithm in abstract terms (for which no compiler or interpreter exists).
2. It describes an algorithm in a clear, but unnecessarily inefficient manner (e.g., an exponential time algorithm describing a problem that can be solved in polynomial time).
3. It describes a non-computable function.

In contrast, we consider a program to be an implementation if it describes an efficient algorithm and is either expressed in a language for which a compiler exists or is expressed in a language that can be directly executed (i.e. machine code) by the targeted processor. Our goal is to define a transformation sequence that, when applied to a specification, will produce an implementation.

It is a violation of the TOP philosophy to apply a transformation rule that produces an incorrect intermediate form that must then be repaired by later transformational steps. This requirement of only producing correct intermediate forms of programs encourages a more global perspective in which software development steps are viewed in relation to the program as a whole. This global perspective allows certain types of complexity to be managed in manner that would otherwise not be possible.

5.1 **Contrasting TOP with Component-based Software Development**

The construction of a jigsaw puzzle provides a useful metaphor for contrasting the differences between TOP development and a more traditional, component-based approach. Figure 4 illustrates how component-based software development might proceed.
Component-based Development

Initially, the system is decomposed into a collection of components taking into account an understanding of the entire system. Interfaces between components are specified. Later, when implementing the interfaces, a component-centric perspective is taken. The focus typically shifts to the interface requirements between components. After the components have been developed, they must then be assembled to produce a software system. The interfaces between components frequently change during the course of the development as requirements change or better abstractions are discovered. When the system design changes in this fashion, great care must be taken to ensure that all interfaces between components are consistent and reflect the current structural decomposition. Assuring such consistency requires a global perspective.

In contrast, TOP development strongly encourages such a global perspective, as shown in Figure 5. Each transformational step produces a program at a lower level of abstraction. The product of each transformational step is a correct program. To refactor or reconfigure interfaces requires that the transformation rules be written with the proper perspective.
5.2 History of TOP

The seeds of transformation-oriented programming can be traced back to the 1970s where a landmark paper by Burstall and Darlington [6] outlined an approach in which a correct but inefficient specification could be transformed (in a semi-rigorous manner) into an efficient, albeit significantly more complex, implementation. Since then, advances in transformation-oriented programming have been sporadic. Research in this area has been triggered by several events such as discovery of new ideas, availability of more powerful computational environments, and increasing demand for highly dependable systems.

Currently, the TOP paradigm is being applied to a wide variety of problem domains within software engineering including synthesis[56], reverse engineering[2][55], and various forms of optimization[52]. The goal in synthesis is to take a program at one level of abstraction and transform it into a program at a lower level of abstraction (where the term “lower” refers to the conceptual distance between a program and its binary executable). In Visser [54], a taxonomy of application areas is given in which transformation goals fall into two broad categories: translation and rephrasing. Translation takes a source program belonging to one language (e.g., specification language) and transforms it into a target program belonging to another language (e.g., a high-level programming language). A rephrasing takes a source program belong to one language and transforms it into a target program belonging to the same language. Given these definitions, synthesis (both refinement and compilation) as well as reverse engineering falls under the category of translation, while optimization falls under the category of rephrasing.
Reverse engineering is essentially the opposite of synthesis. Here the goal is to raise the level of abstraction rather than lower it. Reverse engineering has been used during software maintenance to provide specifications for legacy systems that require new functionality, for example the ability to handle calendar years beyond 1999 (i.e., the Y2K problem).

The goal in optimization is to improve the time and/or space requirements that a program needs in order to execute. Function inlining, variable inlining, and common subexpression elimination are well-known optimization techniques that have been implemented using transformations.

5.3 Transformation Systems General Architecture

A _program transformation system_ is an environment that supports transformation-oriented programming. Such an environment typically includes:

1. a component, such as a parser, capable of defining elements in the domain of discourse (i.e., the programs we are interested in transforming);

2. a specialized _transformation language_ containing appropriate primitives facilitating the development of transformation rules and strategies (We refer to programs written in this transformation language as _transformation programs_. The transformation language typically includes, as primitives, various term traversal operators, some form of matching, and iterators.);

3. an engine for executing transformation programs;

4. a means for displaying the results of transformation steps such as pretty-printing the initial, intermediate, and final forms of a program; and

5. a feedback system, such as a GUI containing a debugger, for facilitating the comprehension of transformation rules and how they are being applied by a particular strategy.

5.4 Syntax and Semantics of Transformation Rules

When designing a transformation system a number of issues must be addressed including the question, “What type of ‘things’ should the system transform?” Transformation systems commonly transform programs, strings, expressions, or even other transformation rules. We consider transformations on two types of structures, abstract syntax trees (ASTs) and syntax derivation trees (SDTs). ASTs can be described by an abstract syntax, and SDTs are described by BNF grammars. When the context is clear, we will refer to these structures as _terms_.

A transformation rule is a variation or extension of a rewrite rule. A rewrite rule consists of a left-hand side, called the _pattern_, and a right-hand side, called the _replace-
A rewrite operator, denoted by the symbol \( \rightarrow \), is used to connect a pattern with a replacement. Thus rewrite rules have the form:

\[
pattern \rightarrow replacement
\]

The purpose of the pattern is to describe a particular term or type of term that one would like to transform. The purpose of the replacement is to define a term that will replace the pattern. Frequently this term is based on some manipulation of the accompanying pattern.

Patterns may contain variables. For example, we may have a rewrite rule of the form:

\[
P(x) \rightarrow Q(b, x)
\]

where \( b \) is a constant and \( x \) is a variable. If we are given the term \( P(10) \), we can match this to the pattern by assigning the term “10” to the variable \( x \). The pair \( x/10 \) is called a substitution. A substitution is applied to a term by replacing variables in the term with the pair of that variable in a substitution. The result of applying the substitution \( x/10 \) to the term \( P(x) \) is \( P(10) \). A set of substitutions is a unifier if, when the substitutions are applied to a pair of terms, the results are identical. When a substitution can be applied to a pattern \( p \) so that the result matches a term \( t \), we say that \( t \) is an instance of \( p \) and that the rewrite rule applies to \( t \).

Rewriting proceeds as follows. For a given term \( t \), a determination is made to see if \( t \) is an instance of some pattern in a rewrite rule. If \( t \) is an instance of a pattern, the rule applies, and a corresponding instance of replacement is constructed by applying the unifier to the replacement. The result of this is used to replace \( t \). In the example above, the result of applying \( P(x) \rightarrow Q(a, x) \) to the term \( P(10) \) is \( Q(a, 10) \).

A pattern defines a set of terms. Determining whether a rewrite rule applies to a term \( t \) involves solving the set membership problem. Unification algorithms [35] can be used to solve the set membership problem. Higher-order, associative-commutative (AC) matching or unification as well as matching and unification modulo equational theories have also been used to further increase the power of patterns to describe sets [11]. In the case of first-order matching, the result of a successful match between a pattern and a term \( t \) is a substitution list in which variables occurring in the pattern are bound to subterms in \( t \). This substitution is then applied to the replacement in order to produce the resultant term (i.e., the term that will be substituted in place of \( t \)). In the case of higher-order matching or AC-matching, the result will typically be a set of substitutions, in which case the result will a set of terms.

Rewrite rules can also be annotated with conditions. In this case, they are called conditional rewrites. A conditional rewrite is typically written as

\[
pattern \rightarrow replacement \text{ if } c
\]

26
where \( c \) denotes a Boolean formula. When a conditional rewrite is applied to a term \( t \), a unification or match is attempted between the pattern and \( t \), if this succeeds, the resulting substitution is applied to \( c \) after which \( c \) is evaluated. If \( c \) evaluates to true, then the conditional rewrite rule produces a transformational step, otherwise the term \( t \) is returned as the result of the rule application.

5.4.1 Example 3

The following example motivates the need for effective strategies for applying transformation rules. Consider a term language defining mathematical expressions. This language allows for integer values and expressions built from addition and multiplication. The following signature defines this language:

**Signature**
- **Sorts:** Integer, Expr
- **Constructors:**
  - **num:** Integer \( \rightarrow \) Expr
  - **plus:** Expr * Expr \( \rightarrow \) Expr
  - **mult:** Expr * Expr \( \rightarrow \) Expr

If we assume first-order matching will be used during rule application, the distribution of multiplication over addition can be described by the following rewrite rules:

*Rule1:* mult (x, add (y, z)) \( \rightarrow \) add (mult (x, y), mult (x, z))  
*Rule2:* mult (add (y, z), x) \( \rightarrow \) add (mult (y, x), mult (z, x))

Another possibility would be the following:

*Rule1'*: mult (x, add (y, z)) \( \rightarrow \) add (mult (x, y), mult (x, z))  
*Rule2'*: mult (x, y) \( \rightarrow \) mult (y, x)

For the first set of rules, any arbitrary exhaustive rule application strategy (such as the fixed-point strategy given in the previous section) will succeed. However, in the second case we must be a little more careful when applying rules. The reason we need to be careful is that *Rule2'* can be applied infinitely often to a term. A possible strategy for this second set of rules would be the following:

1. Apply *Rule1'* to exhaustion
2. Apply *Rule2'* exactly once to every (sub)term
3. If *Rule1'* can be applied to any (sub)term, then goto step 1, else stop.

This example illustrates the difference between rules and strategies. Note that all rules are correctness preserving (in the sense that they preserve the numerical value of the expression) regardless of the strategy that is used to apply them.
6 HATS

The High-Assurance Transformation System (HATS) is a language-independent program transformation system whose development began in the late 1990s at Sandia National Laboratories[57]. The goal of HATS is to provide a transformation-oriented programming environment facilitating the development of transformation rules and strategies whose correctness can be formally verified. The following diagram shows the HATS architecture.

![HATS Architecture Diagram]

In HATS, programs belonging to a particular problem domain are defined by a context-free grammar. Internally, HATS stores and manipulates these programs as syntax derivation trees. HATS provides a special purpose language for defining transformation rules and strategies. This language enables the description of the domain language in the standard manner [1]. A grammar \( G = (V, T, P, S) \) is composed of a set of non-terminal symbols \( V \), a set of terminal symbols \( T \), a set of productions \( P \), and a start symbol \( S \in V \). Each production consists of a non-terminal symbol on the left hand side and a sequence of terminal and non-terminal symbols on the right hand side. A grammar derives a string by beginning with the start symbol and repeatedly replacing a nonterminal by the right hand side of a production whose left hand side matches that non-terminal. Given a context-free grammar \( G \), the notation \( A \xrightarrow{*} \beta \) denotes a derivation belonging to \( G \). The expression \( A[\beta] \) denotes the term whose start symbol (root of the SDT) is \( A \) and whose final form (leaf nodes of the SDT) are \( \beta \).

Presently, an LR(k) parser\(^3\) supporting Extended-BNFs is used to automatically

\(^3\)An LR(k) parser is for a grammar that can be parsed scanning left to right using k symbol look ahead. See Aho [1] pp 215-247 for details.
convert an expression of the form $A[\beta]$ into a completed term. In this framework, nonterminal symbols in the grammar form the basis for defining variables that can be instantiated during matching. In particular, subscripted instances of nonterminal symbols of the context-free grammar are used to denote variables. For example, if $E$ is a nonterminal symbol in our grammar, then $E_1$, $E_2$, $E_3$, ... denote distinct variables of type $E$. Variables are quantified over the sublanguage of terms which they can derive according to the given grammar. As a result, they can only match with a term sharing the same root symbol.

6.1 Writing Transformation Rules

In HATS, a transformation rule, or transform, is a function that is parameterized on the term to which it is applied. At present, HATS uses a first-order matching algorithm to determine if a transform applies to a given term. A distinguishing feature of HATS is that the matching operation is explicit and is denoted by the symbol $==$ . The syntax we will use to describe transformation rules in this chapter is\(^4\)

$$
transform \text{ rule\_name}(\text{variable})\{\text{match\_expression} \to \text{replacement}\}
$$

where $\text{rule\_name}$ denotes the name of the rule, $\text{variable}$ denotes a term variable to which the rule will be applied, $\text{match\_expression}$ is a pattern that when matched returns and instance of $\text{replacement}$. The rule is preceded by the key word $\text{transform}$. The evaluation of a match expression produces a pair consisting of a Boolean value and a substitution list. If the Boolean value is true, the evaluation has succeeded and the substitution is applied to the replacement, which then rewrites the original term. If the Boolean value is false, the transform returns the input term.

6.1.1 Example 4

In this example, we show how the rules for distributing multiplication over addition can be expressed in HATS. Because HATS is SDT-based, we must define our language in terms of a BNF grammar (rather than a signature). The BNF and transformation rules are:

```
factor ::= plus ( factor , factor )
         | mult ( factor , factor )
         | integer

transform Rule1 (factor0)
   { factor0 == factor[ mult( factor1 , plus( factor2 , factor3 ) ) ] }
```

\(^4\)The actual syntax in HATS is somewhat different. For an example of actual HATS syntax, see Winter [57].
\[
\text{factor} \left( \text{plus} \left( \text{mult} \left( \text{factor}_1, \text{factor}_2 \right), \text{mult} \left( \text{factor}_1, \text{factor}_3 \right) \right) \right)
\]

transform Rule2 (factor_0)
{factor_0 == factor \left( \text{mult} \left( \text{plus} \left( \text{factor}_1, \text{factor}_2 \right), \text{factor}_3 \right) \right)
\rightarrow
factor \left( \text{plus} \left( \text{mult} \left( \text{factor}_1, \text{factor}_3 \right), \text{mult} \left( \text{factor}_2, \text{factor}_3 \right) \right) \right)
}

Note that both Rule1 and Rule2 are of the same type in the sense that they can only produce transformational steps when applied to SDT’s having the nonterminal symbol \textit{factor} as the root symbol\textsuperscript{5}. HATS provides the vertical bar symbol, “\text{|}”, as a composition mechanism enabling rules of the same type to be conditionally grouped\textsuperscript{6}. Using the vertical bar, the above rules may be also be written within a single transform as follows:

transform distribute \_ mult1 (factor_0)
{ factor_0 == factor \left( \text{mult} \left( \text{factor}_1, \text{plus} \left( \text{factor}_2, \text{factor}_3 \right) \right) \right)
\rightarrow
factor \left( \text{plus} \left( \text{mult} \left( \text{factor}_1, \text{factor}_2 \right), \text{mult} \left( \text{factor}_1, \text{factor}_3 \right) \right) \right)
}

| factor_0 == factor \left( \text{mult} \left( \text{plus} \left( \text{factor}_1, \text{factor}_2 \right), \text{factor}_3 \right) \right)
\rightarrow
factor \left( \text{plus} \left( \text{mult} \left( \text{factor}_1, \text{factor}_3 \right), \text{mult} \left( \text{factor}_2, \text{factor}_3 \right) \right) \right)
}

Due to the fact that matching is explicit in HATS, match expressions can be elegantly combined with other match expressions or modified through Boolean connectives. This has the effect of embedding application conditions within the match expression, and this gives HATS the ability to express conditional rewrites. Utilizing this feature of HATS enables us to alternately express the distribution of multiplication over addition as follows:

transform distribute \_ mult2 (factor_0)
{ factor_0 == factor \left( \text{mult} \left( \text{factor}_1, \text{plus} \left( \text{factor}_2, \text{factor}_3 \right) \right) \right)
\rightarrow
factor \left( \text{plus} \left( \text{mult} \left( \text{factor}_1, \text{factor}_2 \right), \text{mult} \left( \text{factor}_1, \text{factor}_3 \right) \right) \right)

\text{or}

factor_0 == factor \left( \text{mult} \left( \text{plus} \left( \text{factor}_2, \text{factor}_3 \right), \text{factor}_1 \right) \right)
\rightarrow
factor \left( \text{plus} \left( \text{mult} \left( \text{factor}_1, \text{factor}_2 \right), \text{mult} \left( \text{factor}_1, \text{factor}_3 \right) \right) \right)
}

\textsuperscript{5}In this example the point is moot since the grammar only contains a single nonterminal symbol.

\textsuperscript{6}Technically speaking, the semantics of the vertical bar is similar but not equivalent to the sequential composition of Rules 1 and 2. However, this difference does not come into play in this example.
6.2 Dynamic Transformations

There are situations in which it is not possible to tell a priori what a transformation rule should be, but it is possible to tell the form of the rule. In this section, we describe one of HATS’ more advanced capabilities, rules for creating transformation rules.

When manipulating large terms, two or more data elements that are related with respect to a particular perspective may be scattered within the structure of the term. In the example below, we consider a Java program. We are interested in verifying the correctness of type assignments in the program\(^7\). In the example shown in Figure 7, type information derived from declarations, constants, and expressions and is propagated across assignments.

Consider the code fragment in Form 1 of Figure 7. If we wanted to use a transformational approach to check whether the code fragment is type correct, we might proceed as follows. First, rewrite all occurrences of variables in statements with their declared types and all constants to their types. This would result in the intermediate form in Form 2. Second, apply rules that transform expressions into their types. For example, an expression that adds two integers is of type integer. In this stage, we also introduce a new type called \texttt{ERROR\_VALUE} that is used to denote the type of an expression that is type incorrect. The results of these transformations are displayed in Form 3.

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
int x; & int x; & int x; & int x; \\
bool y; & bool y; & bool y; & bool y; \\
\hline
x = 6; & \texttt{INT}\texttt{=INT}; & \texttt{INT}\texttt{=INT}; & \texttt{INT}=\texttt{INT}; \\
x=x*x; & \texttt{INT}\texttt{=INT}\texttt{\_\_INT}; & \texttt{INT}\texttt{=INT}; & \texttt{INT}=\texttt{INT}; \\
y=\texttt{true}; & \texttt{BOOL}\texttt{=BOOL}; & \texttt{BOOL}\texttt{=BOOL}; & \texttt{BOOL}\texttt{=BOOL}; \\
x=y; & \texttt{INT}\texttt{=BOOL}; & \texttt{INT}=\texttt{BOOL}; & \texttt{ABORT}; \\
y=x \texttt{and } y; & \texttt{BOOL}\texttt{=INT} \texttt{and } \texttt{BOOL}; & \texttt{BOOL}\texttt{=ERROR\_VALUE}; & \texttt{ABORT}; \\
\hline
\end{tabular}
\end{center}

Figure 7: Simple type checking

Finally, construct transformation rules that capture the notion of type correctness at the statement-level. In our example, we define an assignment statement to be type correct if and only if the type of the variable assigned to (i.e., the left-hand side) is identical to the type of the value to be assigned (i.e., the expression on the right-hand side). We introduce a new type of statement called \texttt{ABORT} that we use to denote a statement that is type incorrect. Form 4 shows the results of applying these transformations.

Performing simple type checking on programs in a transformational manner is conceptually straightforward. The major problem here is how the type of an identifier, which is found in the declaration, can be distributed throughout the program. We will refer to this problem as the distributed data problem. The root of the distributed data

\footnote{This example was inspired by a similar example given by Visser [53].}
problem lies in the fact that one part of the program (e.g., the occurrence of an identifier in an expression) needs information that occurs in another part of the program. The syntactic distance between the declaration of an identifier and its first use can be arbitrarily large. This makes it impossible to use standard first-order matching to capture, within a single match, the declaration of an identifier and its first use\(^8\). Somehow we need to achieve the transmission of data between transformation rules.

In the example above, we simply wrote transformation rules that rewrote all occurrences of \(x\) and \(y\) to the types \(\text{INT}\) and \(\text{BOOL}\) respectively. That is, we “hardwired” the type relationships within the transformation rules. These transformation rules are straightforward, and their correctness is easily shown. However, our goal is to write a general set of rules that is capable of type checking all programs. The distributed data problem arises precisely because we do not know the declarations that are in the code fragment at the time that we are writing our transformation rules.

HATS makes use of meta-transformation rules called *dynamic transformation rules*. A dynamic transformation rule takes a term as input and returns a set of transformation rules as its result. In contrast, a transformation rule takes a term as input and returns a term as its output. In HATS, the basic syntax of a dynamic transformation rule is:

```plaintext
dynamic_rule_name (variable)
{ match_expression
    \rightarrow
    transform (variable) {match_expression \rightarrow term}
}
```

Let us examine how the distributed data problem can be solved using dynamic transformations. At this point, we include the BNF grammar describing our mini-programming language.

\(^8\)If we contemplate extending the capabilities of our matching algorithm, we run into other problems like nested scopes containing multiple declarations of the same identifier. Another possibility is to somehow “carry” the information found in the declaration to the places in the program where the information is needed. In an imperative or functional paradigm, parameter passing is a key mechanism that is used to transmit information. Global variables can also be used, but it is widely accepted that these should be used sparingly as the concept of a global variable violates referential transparency, making it more difficult to reason about and maintain the program.
Figure 8: BNF Grammar of mini-programming language

| prog | ::= | decl_list ; stmt_list ; |
| decl_list | ::= | decl ; decl_list | decl |
| decl | ::= | type id |
| stmt_list | ::= | stmt ; stmt_list | stmt |
| stmt | ::= | assign | ABORT |
| assign | ::= | type_term = expr |
| expr | ::= | expr b_op term | term |
| term | ::= | boolean | ( expr ) | not(expr) | E |
| b_op | ::= | or | and |
| E | ::= | E + T | T |
| T | ::= | T * F | F |
| F | ::= | ( expr ) | int | type_term |
| type_term | ::= | type | id |
| id | ::= | ident |
| type | ::= | type_name |
| int | ::= | integer_value |
| boolean | ::= | boolean_value |

Given the grammar in Figure 8, a dynamic transformation that solves the distributed data problem can be written in HATS as follows:

dynamic distribute_type_info (decl0)
\{ decl0 == decl[ type1 id1 ] \}
    ->
    transform (type_term0)
      \{ type_term0 == type_term[ id1 ] \rightarrow type_term[type1] \}

When this dynamic transformation is applied to our sample program, it will produce the following sequence of (anonymous) transformation rules:

transform (type_term0) \{ type_term0 == type_term[ x ] \rightarrow type_term[ int ] \}
transform (type_term0) \{ type_term0 == type_term[ y ] \rightarrow type_term[ bool ] \}

Given these rules, we can easily transmit the type information of an identifier to places in the program where it is needed (e.g., a fixed point application of the above transformation rules to the statement list portion of the program). Furthermore, given the semantics of dynamic transformations, it is easy to conclude that the distributed data problem has been solved correctly.
7 Embedded Systems

Advances in computer technology are rapidly increasing the computational power that can be brought to bear within embedded systems. Chip designs are now reaching the point where the computational power they provide makes it possible to include dependability as a primary design objective, which in turn has opened the door for the consideration of such systems in high-consequence applications. The bottleneck facing high-consequence embedded system developers is their ability to provide sufficient evidence that an embedded design satisfies a given set of stringent dependability requirements.

Embedded systems are being developed to solve increasingly complex problems. For practical reasons, designers must give serious consideration to incorporating or leveraging existing research results and technologies such as robust programming languages and COTS products. When this is done judiciously, it becomes possible to develop dependable systems within reasonable time frames and cost constraints.

In this section we consider the adaptation of the Java Virtual Machine (JVM) to a particular class of embedded system designs. The specification of the JVM provides it with the ability to dynamically load and link classfiles during execution. This feature enables an application to begin execution before all of its classfiles have been loaded. Such eager execution is highly beneficial for applications having classfiles distributed across the internet. However, the price that must be paid for this functionality is that many attributes of an application which are typically considered to be static (i.e., they can be resolved at compile-time) are now dynamic (i.e., their resolution must occur at runtime).

In many embedded applications, the eager execution capability provided by the JVM is not useful or even desirable. For example, downloading classfiles over the internet during execution may present an unacceptable risk for a high-consequence application. Thus, the design for such an embedded system allows for the (more traditional) separation of the static and dynamic aspects of the JVM. As shown in Figure 9, the goal of the classloader (the static portion of the JVM) is to take a set of classfiles as input and output an intermediate form in which all static aspects of the classfiles have be resolved.

Because the goal of the classloader is to manipulate a formally defined input (i.e., a set of classfiles), the problem is an ideal candidate for TOP. Furthermore, due to the size of the input space, testing, when used exclusively, is not an effective method for providing strong evidence in the correctness of the classloader. Thus, in order to achieve high-assurance other forms of evidence must be provided.
In this section, we demonstrate how program transformation can be used to positively impact an embedded system design having the architecture described above. In particular, we describe how the static functionality of the Java Virtual Machine (JVM) can be realized through TOP.

7.1 The SSP Project

At Sandia National Laboratories, an effort is underway to develop a system called the Sandia Secure Processor (SSP). This system consists of the SSP-classloader function that is implemented in software and the SSP-runtime function that is implemented in hardware. The intent is to develop a general-purpose computational infrastructure suitable for use in high-consequence embedded systems. Because of this, considerable resources are dedicated to providing strong evidence that all aspects of the SSP, both the classloader and the runtime, have been designed and implemented correctly.

In typical stand-alone embedded applications, all classes can be made available to the JVM before there is a need to invoke the application’s main method. Because of this, loading, verification, most of preparation, and resolution can be done statically. The SSP has been suitably designed to enable a clean separation of the static and dynamic aspects of the preparation step thereby allowing the static portion to be shifted to the classloader and the dynamic portion to be shifted to the runtime.

The SSP is based on the JVM with three significant restrictions. First, threads are not supported. Second, strings and real numbers are not supported. Finally, dynamic loading is not supported. The classloader completes execution before the SSP runtime begins. This separation allows the development of a microprocessor implementing only

35
the runtime function of the SSP, which results in a reduction of the size of the microprocessor gate count, a reduction in microprocessor complexity, and an increase in execution speed.

7.2 The SSP-classloader

The specification of the JVM states that classes should be made available to a running program via a loading and linking sequence. The loading step consists of importing a classfile into the JVM. A classfile is the binary form of a class and is typically generated by a Java compiler. The linking phase can be broken down into three steps: (1) verification, (2) preparation, and (3) resolution. Verification ensures that a class is well formed. Preparation involves allocating the appropriate amounts of memory for classes and objects and assigning default values to these memory locations. Resolution is the act of transforming symbolic references into direct references with respect to a given hardware architecture.

The job of the SSP-classloader is to correctly translate Java classfiles into a form suitable for execution by the SSP-runtime. This translation produces an intermediate form that we call a ROM image, and concerns itself with many issues such as:

1. Resolving symbolic references to physical addresses or direct and indirect offsets.
   This resolution is intimately linked to the architecture of the hardware.

2. Correctly capturing the inheritance semantics of Java for applications consisting of multiple classfiles.

3. Providing suitable information for method invocation and return.


---

9One of the unique aspects of this project has been the close relationship between the classloader development team and the SSP development team. Both teams have frequent contact and are willing to negotiate design complexity issues in favor of increasing the analyzability of the overall system.
The correctness of the classloader is the attribute that is of primary importance. Other attributes such as spatial and temporal efficiency are a distant second. Given these constraints, program transformation provides a technology well suited for realizing the functionality of the classloader in an offline manner. While it is true that a more efficient implementation of the classloader can be constructed, for example by simply implementing the SSP-classloader in C, the verification of such a classloader would undoubtedly be intractable. In contrast, program transformation lays the groundwork that provides the possibility of formally verifying the correctness of the classloader.

### 7.2.1 The SSP-runtime

The SSP-runtime is the hardware component of the SSP. It is charged with the responsibility of correctly executing bytecodes as well as handling any runtime requirements, such as exceptions, that result from their execution. The execution of the SSP with a given ROM image should produce the same behavior as the execution of a correct JVM implementation given the corresponding set of class files. This can be stated more precisely as follows:

- Let \( App \) denote the set of classfiles in a Java application.
- Let \( SSP\text{-classloader} \) and \( SSP\text{-runtime} \) denote the classloader and runtime functions of the SSP.
- Let \( JVM \) denote a Java Virtual Machine.
- Let \( JVM(App) \) denote the execution of \( App \) by \( JVM \).
• Then $SSP_{-}runtime(SSP_{-}classloader(App)) = JVM(App)$

Below is a partial list of requirements the SSP-runtime design should satisfy:

1. There must be the option of building the processor using radiation-hardened technology for military applications.

2. An open-source for the system must be available allowing detailed design analysis and testing of all aspects of the system down to the gate level.

3. Certification evidence should be provided by formal mathematical proofs of correctness to the extent possible, and strongly convincing evidence must be provided in all other cases where mathematical proofs have not been achieved.

4. A security policy must be strictly enforced ensuring that any program is either rejected as incorrect by compile-time or run-time checks, or its behavior must be understandable by reasoning based entirely on the language semantics, independent of the implementation. In particular, no violation of this policy may be permitted regardless of whether it results from an inadvertent error or a malevolent attack.

5. The processor should support an I/O interface such that the impact of I/O on computation is separated to as large an extent as possible. For example, I/O implementation via interrupts may dramatically impact computation time and is thus a poor design choice.

6. The processor should be compatible with Java technology (e.g. development environments, compilers, and byte-code verifiers).

At Sandia National Laboratories, an initial version of the SSP-runtime has been designed in VHDL. All indications are that it will be less than 40K gates and capable of operating near 75MHz. Most importantly however is that the simplifications have reduced by at least an order of magnitude the number of test vectors required to fully verify the correct operation of the digital logic.

Current plans for fabrication are to use the design as a processor core in a system-on-a-chip that is based on a previously designed system. The system, shown in Figure 11, was fabricated in CMOS6 technology and used a core based on Intel’s 8031. The total number of gates in that system was approximately 120K gates, of which the processor core consumed roughly 25K gates. The total die size was 378 x 347 mils, and was packaged in a 208 BGA ceramic package.
The core logic of the new VHDL design synthesizes to approximately 37K gates. The radiation hardened CMOS6R process will be used for initial fabrication. Tuning it for the desired radiation hardness characteristics in that technology will allow a maximum clock rate of 35Mhz. It is still unclear the number of bond pads that will be required as the testing methodology for the die is still being developed. It is expected that enhancements made to the surrounding logic coupled with a number of static memories to be included on the die would yield a final equivalent number of gates that approaches 200K gates.

A second-generation part is intended to target the Silicon On Insulator (SOI) technology. Its smaller geometry will allow for the resultant maximum clock rate of 50Mhz.

7.3 A Transformation-based Classloader

In this section we describe a TOP solution to the classloader problem. As a precondition we assume that classfiles can be represented by terms according to a given grammar description. The structure of these classfile terms closely matches the structure of classfiles as described in the JVM specification [30]. However, there are a few differences: (1) constant pool entries are explicitly indexed, (2) constant pool entries are modified so that they have a more homogeneous structure, and (3) constant pool indexes throughout the classfile are also correspondingly altered except for indexes that occur within bytecodes (e.g., checkcast, getfield). These alterations will be discussed in more detail in the following sections. The important point to remember is that the classfile is translated to a term (i.e., a parse tree) whose structure is similar, but not identical, to the term structure described in the JVM specification.
Under these conditions, the goal is to develop a set of transformation rules that when guided by an appropriate strategy will transform a collection of classfiles belonging to a Java application into a ROM image. A prototype transformation-based classloader has been implemented and the lessons learned have been used to extend and enhance the capabilities of our transformation system.

It is difficult to construct a suitable term language that describes the structure of Java classfiles, the structure of the ROM Image, and necessary intermediate forms. The capabilities of the transformation system (e.g. higher-order matching, AC-unification, dynamic transformations) as well as the transformational approach envisioned by the development team affect the design of the intermediate forms needed. There are many aspects that come into play here such as how the system matches terms, constructs replacement terms, as well as controls the application of transformation rules. One of the areas that we are actively researching is the application of various software engineering techniques such as UML class diagrams and OO analysis to help us in the construction of term languages [12].

7.3.1 Constant Pool Resolution

We will use the term resolution to refer to the various translations needed to convert a set of classfiles into a ROM image. Classfiles are assumed to come from trusted sources and are loaded offline. Thus, much of the information contained in a classfile can be discarded. The ROM image of a classfile consists predominantly of a constant pool and a method areas.

The heart of resolution performed by the classloader is the construction of the ROM image constant pool. For a given classfile, constant pool resolution is accomplished by transforming the classfile through a sequence of canonical forms, each of which has some abstract property that can be utilized by a later transformational stage. The following canonical forms are used.

**Canonical Form 1: Removal of Indirection.** A useful intermediate form when creating a ROM image is to remove all indirection in the constant pool. Initially, various types of information within a Java classfile are expressed as indexes into the constant pool. For example, bytecodes within methods can have constant pool indexes as arguments, and the name and type of a field element are expressed in terms of constant pool indexes. Furthermore, constant pools themselves store much of their information internally through indirection. For example, a class element entry contains a name_index rather than the name of the class. The constant pool entry at name_index is a string (i.e., a constant_utf8_info entry) whose value is the name of the class.

**Canonical Form 2: Relevant Constant Pool Construction.** Since the SSP does not support Java string types, the string entries serve no useful purpose. Thus, constant_utf8_info and constant_name_and_type_info entries can be removed. We call the form that remains the relevant constant pool.
Canonical Form 3: Offset Indexing. In this form, an offset is computed for each relevant constant pool entry. The reason for this is that the SSP accesses constant pool entries via offsets rather than abstract indexes. In contrast, Java classfiles access constant pool entries via abstract indexes. The difference between an offset and an abstract index is that an offset allows entries to have varying sizes (e.g., a 2 word entry versus a 1 word entry) while an abstract index considers all entries to have equal size.

Canonical Form 4: Resolution of Fields. All field entries must be resolved either to absolute addresses (for static fields) or to offsets (for instance fields). Both of these resolutions require global information. For example, the absolute address of a static field depends on how many such addresses have been assigned during the resolution of other constant pools. The assignment of offsets to instance fields requires knowledge of where within the inheritance hierarchy this field declaration falls.

Canonical Form 5: Resolution of Classes. Resolved class entries consist of an address to the class’s method area structure and the object size for that class (e.g., how many instance fields an object will have).

Canonical Form 6: Resolution of Methods. Java supports three types of methods: virtual, static, and special. Within the ROM image, resolved constant pool entries for static and special methods occupy two words of memory and contain the following information: (1) the memory size of the method’s local variables, (2) the memory size of the methods parameters, (3) the location of the start of the method’s bytecodes, and (4) the address of the constant pool structure associated with the method. In this chapter, we make the simplification that a virtual method element also occupies two words of memory. In the actual ROM image for the SSP, a virtual method occupies only one word.

The following sections describe in detail how transformations are used to realize the constant pool resolution steps for removal of indirection, construction of a relevant constant pool, and offset indexing.

7.4 Removal of Indirection

In this section we begin with a concrete example that shows how indirection can be removed from the entries in a constant pool that has been modified to improve its readability by humans. (See appendix A, B, and C for a more complete example). With respect to an SSP constant pool, we have taken some liberties with how we present constant pool data. For example, we assume that abstract indexes for the constant pool have been made explicit.

Consider the following constant pool entries describing the field \( x \) of type integer belonging the to the class \textit{animal}, shown in Figure 12. This constant pool has six entries. Entry 4 in the table that describes \( x \). This entry contains two pieces of information:
the class_index, which leads to the class, and the name_and_type_index, which leads to the name and type of the variable. The class_index refers to the second entry in the constant pool. This entry contains the index of a constant pool entry that contains a class name, namely the first entry. The first entry is a constant string of type constant_utf8_info and contains the string animal. The name_and_type_index of entry 4, through double indirection, arrive at the variable name x and the type I, contained in the final two table entries shown here.

<table>
<thead>
<tr>
<th>Index</th>
<th>Original Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant_utf8_info</td>
<td>animal</td>
</tr>
<tr>
<td>2</td>
<td>constant_class_info</td>
<td>name_index = 1</td>
</tr>
<tr>
<td>3</td>
<td>constant_name_and_type_info</td>
<td>name_index = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>descriptor_index = 6</td>
</tr>
<tr>
<td>4</td>
<td>constant_fieldref_info</td>
<td>class_index = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>name_and_type_index = 3</td>
</tr>
<tr>
<td>5</td>
<td>constant_utf8_info</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>constant_utf8_info</td>
<td>I</td>
</tr>
</tbody>
</table>

Figure 12: Unresolved constant pool entries

The first task is to remove one level of indirection in this table. For example, we can take the name_index in the second entry and replace it with the value animal. The type of the entry in the constant pool changes. For example, the second entry should have type constant_utf8_info after the index is replaced by the string value. In the tables below, we keep the original type names and infer the actual type from the context. The removal of one level of indirection from each constant pool entry yields the table shown in Figure 13.

<table>
<thead>
<tr>
<th>Index</th>
<th>Original Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant_utf8_info</td>
<td>animal</td>
</tr>
<tr>
<td>2</td>
<td>constant_class_info</td>
<td>animal</td>
</tr>
<tr>
<td>3</td>
<td>constant_name_and_type_info</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>constant_fieldref_info</td>
<td>class_index = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>name_index = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>descriptor_index = 6</td>
</tr>
<tr>
<td>5</td>
<td>constant_utf8_info</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>constant_utf8_info</td>
<td>I</td>
</tr>
</tbody>
</table>

Figure 13: Partially resolved constant pool entries

The removal of one more level of indirection yields the table shown in Figure 14.
<table>
<thead>
<tr>
<th>Index</th>
<th>Original Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant_utf8_info</td>
<td>animal</td>
</tr>
<tr>
<td>2</td>
<td>constant_class_info</td>
<td>animal</td>
</tr>
<tr>
<td>3</td>
<td>constant_name_and_type_info</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>constant_fieldref_info</td>
<td>animal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>5</td>
<td>constant_utf8_info</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>constant_utf8_info</td>
<td>I</td>
</tr>
</tbody>
</table>

Figure 14: Resolved constant pool entries

In order to put a class file in Canonical Form 1, we need to construct transformation rules that perform the operations shown above. We begin by considering how the removal of indirection can be expressed for constant-pool-like structures where each entry is either a data element or an index. Let \( CP \) denote an array corresponding to our constant pool. Abstractly, the removal of one level of indirection from an entry \( k \) would be achieved by simply replacing the current value of entry \( k \) (say, an index) with the entry in \( CP \) that is indexed by entry \( k \). Transformationally this concept can be expressed in a rewrite rule-like style as:

\[
CP[k] \rightarrow CP[CP[k]] \quad \text{when} \quad CP[k] \text{ is an index}
\]

More concretely, let us assume that \( CP \) is denoted by a list of tuples of the form: \((index, data)\), where the first element of the tuple denotes the position of the tuple in the list, and data may be an index or a string (e.g. a utf8). For example, suppose we were given a constant pool of called \( CP \) with the following form:

- \((1, 3)\)
- \((2, 4)\)
- \((3, “hello”)\)
- \((4, “world”)\)

The following dynamic transformation rule captures the removal of indirection concept.

dynamic remove_indirection (entry1) 
{entry1 == entry[(index1, data1)]} 
\rightarrow 
transform (entry2) {entry2 == entry[(index2,index1)] \rightarrow entry[(index2, data1)]}

43
This dynamic transformation rule, when instantiated with respect to a particular value for \( entry_1 \), will create a transformation rule that rewrites indirect references to \( entry_1 \) with the actual data corresponding to the reference. If we apply the above dynamic transformation rule to \( CP \), the system will (internally) generate the following set of transformation rules, one rule for each entry in \( CP \):

\[
\begin{align*}
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,1) ]} \} \\
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,2) ]} \} \\
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,3) ]} \} \\
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,4) ]} \} \\
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,5) ]} \} \\
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,6) ]} \}
\end{align*}
\]

If we now apply the above transformation rules to \( CP \) we will get:

1. “hello”
2. “world”
3. “hello”
4. “world”

Because the first two transformation rules above are never applied, one might conclude that they are not needed. In general, this is not the case. The reason the rules did not apply is because the level of indirection in the example given was only one level deep. Consider the addition of \( (5,1) \) and \( (6,2) \) to \( CP \). The dynamic transformation rule generates two additional rules:

\[
\begin{align*}
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,5) ]} \} \\
\text{transform (entry2)} & \{ \text{entry2} \rightarrow \text{entry[ (index2,6) ]} \}
\end{align*}
\]

When this set of six rules is applied to the new \( CP \), the last two rules are not used, but the first two are used. In general, it is not possible to determine in advance which rules will be applied.

This example demonstrates how the removal of indirection might be realized in a transformational manner. In order for this idea to be applied to the removal-of-indirection problem in the SSP classloader, a term language supporting suitable intermediate classfile forms must be designed. Below is an example of a partially completed context-free grammar describing such a term language. The grammar fragment below is intentionally ambiguous. HATS can parse such grammars so long as the transformation rules themselves do not define any terms in an ambiguous manner.

```
ClassFile ::= magic
minor_version
major_version
constant_pool_count
constant_pool
```
access_flags
this_class
super_class
interface_count
interfaces
fields_count
fields
methods_count
methods
attributes_count
attributes

constant_pool ::= cp_info_list
cp_info_list ::= cp_info cp_info_list | () the () denotes the ε-production
cp_info ::= access base_entry | base_entry
access ::= index | offset index

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

constant_name_and_type_info ::= info2
constant_fieldref_info ::= info2
constant_methodref_info ::= info2
constant_class_info ::= info1
constant_integer_info ::= bytes
constant_utf8_info ::= info1

info1 ::= index | utf8
info2 ::= index | class name_and_type | name descriptor
class ::= name
name ::= info1
name_and_type ::= info2
descriptor ::= info1

This is similar to the structure of class files specified in the JVM specification [30]. The major differences are:

1. a constant pool entry can be a base entry, or it can be a base entry preceded by access data (e.g., an index);
2. access data can be an abstract index (as defined by in JVM specification) or an offset address and an abstract index;

3. entities such as name, class, and descriptor are of type info1 and the name_and_type entry is of type info2;

4. info1 can be either an index or a utf8 value; and

5. info2 can be an (a) index, (b) a class and name_type, or (c) a name and descriptor.

While the construction of the term language shown above is not particularly complex, the structure and abstractions of such a grammar can profoundly influence how the developer thinks and expresses transformational ideas, both at the rule level and the strategy level. Grammars define how information can be represented. The nonterminal symbols in a well-designed grammar describe important semantic concepts in the language. Similarly, important relationships and groupings between concepts can also be captured. We believe that it is well worth the effort to construct grammars in which the right semantic relationships are elegantly captured at the syntactic level.

In order to help us design a transformation strategy for removing indirection, let us look at the structure of some constant pool entries. Figures 15 and 16 show the parse trees (i.e., terms) that describe five types of constant pool entries: utf8 entries, class entries, field entries, method entries, and name_and_type entries.

![Parse trees for constant pool entries](image-url)

Figure 15: Parse trees for constant pool entries

46
Figure 16: Detailed parse tree of dual index entry

Each constant pool entry type is ultimately defined with respect to either an info1 or an info2 element. This is an important structural design decision that permits writing transformation rules that rewrite info1 and info2. To see how this might be done, let us now look more closely at the structure of info1 and info2 shown in Figure 17.

Figure 17: The structure of info1 and info2.

As indicated by the grammar, info1 can be either an index or a utf8 value.
Similarly, info2 can be an index, a name and descriptor, or a class followed a name_and_type value. In turn, name, descriptor, classes, and name_and_type are all defined in terms of info1. Given these structural relationships, we can resolve indirection by: (1) rewriting all info2 index terms to an info2 name and descriptor term or to an info2 class and name_and_type term; and (2) rewriting all info1 index terms to info1 utf8 terms. Such rewriting will remove all info1 and info2 indexes.

Figures 18 and 19 show how this approach would resolve a constant_methodref_info element.

**Figure 18: Resolution of info2.**
Figure 19: Resolution of info1.

We need to construct transformation rules that rewrite info2 and info1 indexes to their proper non-index terms. Notice that a class index references a class entry, and a class entry is an index to a utf8 element. Thus, we have two levels of indirection in this case. The dynamic transformation rule shown below constructs transformation rules that resolve both types of indirection occurring in info1 terms. This dynamic transformation also constructs transformation rules that resolve the indirection that arises in info2 terms.

dynamic resolve_info (cp_info0)
{ (* this section constructs transformation rules for info1 indexes *)
  (cp_info0 == cp_info[ index1 constant_utf8_info1 ]
   and
   constant_utf8_info1 == constant_utf8_info[ info1 ]
  )
  or
  (cp_info0 == cp_info[ index1 constant_class_info1 ]
   and
   constant_class_info1 == constant_class_info[ info1 ]
  )
  →
  transform (info10) {info10 == info[ index1 ] → info11 }

  (* this section constructs transformation rules for info2 indexes *)

49
cp_info0 == cp_info[ index1 constant_name_and_type_info1 ]
and
constant_name_and_type_info1 == constant_name_and_type_info[ info21 ]
→
transform (info20) { info20 == info2[ index1 | → info21 ] }
}

A feature of HATS seen in the above transformation rule is that match expressions can be disjunctions of other match expressions. Disjunctions come into play when information is common across the replacement of a number of transformation rules. In such cases, match expressions can be factored out of multiple rules and formed into a disjunction. For example, consider n rules having distinct match expressions but producing the same replacement term. Without the ability to express disjunction one would have to write n distinct rules:

Rule 1: match_expr_1 → X
Rule 2: match_expr_2 → X
...
Rule n: match_expr_n → X

However, with disjunction, these rules can be factored into a single rule:

Rule 1: match_expr_1 or match_expr_2 or ... or match_expr_n → X

The dynamic transformation, resolve_info, when applied to a classfile, will create the set of transformation rules, T1, needed for removal of indirection. What remains is for these rules to be applied by an appropriate strategy. Recall that our goal is to remove all indirection (i.e., all indexes) through the entire classfile. Obviously a single pass is not sufficient; thus, an exhaustive application strategy is used. The transformation rule shown below defines a strategy that is capable of removing the indirection within classfiles.

In the resolve strategy, the expression eval(post_order, resolve_info, ClassFile0) applies the dynamic rule resolve_info in order to produce a set of transformation rules which can then be applied to the entire classfile in order to remove indirection.

The transformation rule below defines a strategy for applying the resolve rule.

transform resolve (ClassFile0)
{ ClassFile1 == fix(post_order, ClassFile0, eval(post_order, resolve_info, ClassFile0))
 →
   ClassFile1
}

50
SIDEBAR
The current version of HATS provides three universal iteration operators: first, once, and fix. All iterators take three arguments: (1) the order that a term is to be traversed (pre-order or post-order), (2) the term to be traversed, and (3) the transformation rules to be attempted at every point in the traversal. The first operator traverses the term in the order specified and continues until the transformation rule is successfully applied for the first time. If the rule does not apply at all then when first reaches the end of its traversal it returns the original term (i.e., it leaves the input term unaltered). The once operator traverses the term in the order specified and attempts to apply the transformation rule to every point in the term. When the end of the traversal is reached, the resulting term is returned. And finally, the fix operator continues to traverse the term in the order specified until the transformation rule can no longer be applied. Note that this may require multiple traversals and can continue indefinitely in cases where the transformation rule is non-terminating.

Presently, the concept of a strategy and how it can be defined within HATS is undergoing a major revision, both at the theoretical level as well as the notational level. Our intention is to provide the user with more general capabilities for defining general strategies such as the recursive closures in Stratego as well as some of the concepts defined in the Rho-calculus.

It is worth mentioning that constant pool indexes throughout the entire classfile will be resolved using the above strategy. With the exception of indexes that occur within bytecodes (e.g., checkcast, getfield), a global resolution is exactly what our design calls for. In the approach presented, index resolution will not apply within the context of bytecodes because terms describing indexes within bytecodes differ from terms describing indexes in the remainder of the classfile. This restriction was stated at the beginning of the section. It is relatively easy to relax this restriction and require the selection of index resolution to be controlled by the strategy.

7.5 Relevant Constant Pool Construction
Continuing on with our example, we now show the relevant constant pool can be extracted from the present form of the constant pool. Recall that the constant pool with indirection removed has the form shown in Figure 20.
<table>
<thead>
<tr>
<th>Index</th>
<th>Original Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant_utf8_info</td>
<td>animal</td>
</tr>
<tr>
<td>2</td>
<td>constant_class_info</td>
<td>animal</td>
</tr>
<tr>
<td>3</td>
<td>constant_name_and_type_info</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>constant_fieldref_info</td>
<td>animal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>constant_utf8_info</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>constant_utf8_info</td>
<td>I</td>
</tr>
</tbody>
</table>

Figure 20: Fully populated constant pool.

The constant pool with only relevant entries is shown in Figure 21.

<table>
<thead>
<tr>
<th>Index</th>
<th>Original Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant_class_info</td>
<td>animal</td>
</tr>
<tr>
<td>3</td>
<td>constant_fieldref_info</td>
<td>animal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 21: Relevant entries after resolution.

An important observation at this point is that the abstract indexes (i.e., 2 and 4) no longer correspond to the position of the entry in the constant pool. This however does not present a problem because at the beginning of this example the abstract indexes were made explicit (i.e., part of the constant pool entry).

Here, the goal is to discard all `constant_utf8_info` and `constant_name_and_type_info` entries. The transformation given below applies to all constant pool lists in which the first element is either a `constant_utf8_info` or `constant_name_and_type_info`. A successful application of the transformation will result in this first element being dropped from the list. A single traversal of the constant pool will produce the relevant constant pool. Because we have defined a constant pool entry list (i.e., `cp_info_list`) as a list whose last element is the empty string, we know that the final element of a `cp_info_list` will always be the empty string and therefore will not need to be removed.

```
transform relevant_cp (cp_info_list_0)
{ cp_info_list_0 == cp_info_list[ index1 constant_utf8_info1 cp_info_list1 ]
  or cp_info_list_0 == cp_info_list[ index1 constant_name_and_type_info1 cp_info_list1]
  ->

52
```
transform relevant (ClassFile0)
{ ClassFile1 == once(post_order, ClassFile0, relevant_cp) → ClassFile1 }

### 7.6 Offset Indexing

We continue with our example showing the result of adding offset information to our relevant constant pool. In this example, the offset for the first relevant constant pool entry is 4 words (i.e., the words at offset 0 – 3 in the constant pool are reserved for some other information). Furthermore, we are given that the class entry has a size of one word; hence the `constant_fieldref_info` entry has an offset of 5 words as shown in Figure 22.

<table>
<thead>
<tr>
<th>Offset Index</th>
<th>Original Index</th>
<th>Original Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004</td>
<td>2</td>
<td><code>constant_class_info</code></td>
<td>animal</td>
</tr>
<tr>
<td>0005</td>
<td>4</td>
<td><code>constant_fieldref_info</code></td>
<td>animal x I</td>
</tr>
</tbody>
</table>

Figure 22: Relevant constant pool entries with physical offsets.

Expressing how offsets should be inserted into constant pool entries is simple, though the notation is somewhat verbose. Our approach is to define two transformation rules that accomplish simple rewrites and a third, strategic rule that controls their application. The `initial_offset` rule inserts, in the first element of the constant pool, an initial offset whose value is obtained from the function call `baseOffset()`. The value of this base offset is defined by the specification of the structure of the ROM image. The `percolate_offset` rule adjusts and “percolates” the offset through the remaining entries in the constant pool according to the size of each entry.

transform `initial_offset` (cp_info0)
{ cp_info0 == cp_info[ index1 base_entry1 ]
  and
  (* the function baseOffset is a library function *)
  { offset1 := baseOffset() } }
  →
  cp_info[ offset1 index1 base_entry1 ]
}

Note that the `initial_offset` transformation makes use of a library call to the function `baseOffset` denoted by the expression:

{offset1 := baseOffset()}

53
Such functionality, supported by HATS, falls outside the realm of pure rewrite-based transformation. Nevertheless, it can be successfully applied in cases where one is willing to assume that the library function used is correct. A classic example where such functionality is highly beneficial is in situations (such as the upcoming transformation) where one wants to perform mathematical operations (e.g., addition and division). While it is possible to write transformation rules capable of performing such mathematical operations, doing this does not increase the understandability or correctness of the program.

The second transformation rule, percolate_offset, applies to adjacent constant pool entries where the first entry has an offset and the second entry does not. In such a situation, the offset for the second entry should be the value of the first offset plus 1 or 2 depending on the type of the first base_entry. If the first base_entry denotes a class, an integer, or a fieldref, then the offset for the second entry is equal to the offset for the first entry plus 1; otherwise, it is equal to the offset for the first entry plus 2.

transform percolate_offset (cp_info_list0)
{ cp_info_list0 == cp_info_list[ offset1 index1 base_entry1
  index2 base_entry2 cp_info_list1 ]
and
  ( base_entry1 == base_entry[ constant_class_info1 ] or
    base_entry1 == base_entry[ constant_integer_info1 ] or
    base_entry1 == base_entry[ constant_fieldref_info1 ])
and
  (* the function u4_plus1 is a library function *)
  { offset2 := u4_plus1(offset1) }
→
  cp_info_list[ offset1 index1 base_entry1
    offset2 index2 base_entry2 cp_info_list1 ]
| (* conclude base_entry1 is a 2 word entry *)
  cp_info_list0 == cp_info_list[ offset1 index1 base_entry1
    index2 base_entry2 cp_info_list1 ]
and
  (* the function u4_plus2 is a library function *)
  { offset2 := u4_plus2(offset1) }
→
  cp_info_list[ offset1 index1 base_entry1
    offset2 index2 base_entry2 cp_info_list1 ]
}
above transformation rules is controlled by the strategy `calculate_offsets`. In this rule, the word `first` indicates that the rule is to be applied to the first match only. After one application succeeds, HATS discontinues its attempt to apply the rule. The word `once` indicates that HATS will traverse the entire term one time, applying the transformation everywhere it matches.

```plaintext
transform calculate_offsets (cp_info_list0)
  { cp_info_list1 == first(post_order, cp_info_list0, initial_offset)
  and
  cp_info_list2 == once(pre_order, cp_info_list1, percolate_offset)
  ->
  cp_info_list2
  }
```

8 Future Work: Verification

Earlier in this chapter we mentioned that one of the attractive features of TOP is that one can provide strong evidence that the output term of a transformation sequence is correct with respect to its input term. The ideal situation arises when transformation rules and strategies are so simple that the correctness of all rules and strategies can be (1) informally validated in situations where validation is considered to constitute acceptably strong evidence, or (2) automatically verified using theorem provers in situations where mathematical certainty constitutes strong evidence. In this section we discuss current work towards verification of the SSP classloader. It is our intent to complete the formal verification of the classloader in the near future.

In a TOP paradigm, problems are solved by transformation sequences, which are typically constructed in the following manner. First, an overall `transformation-oriented design` (TOP design) is developed in which a number of canonical forms are identified. Then an ordering of these canonical forms is determined. Finally, transformation rules and strategies are developed. We use the term `TOP implementation` when referring to the transformation rules and strategies that implement a TOP design. Because of the influence the structure of the term language can have over a TOP implementation, it is generally advisable to develop the term language concurrently (or iteratively) with the transformation rules and strategies when possible.

The two objectives of verification in a TOP framework are design verification and implementation verification. The objective of design verification is to show that passing an input term through the various canonical forms identified in the design produces an output term that solves the problem. The objective of implementation verification is to show that the design is correctly implemented, i.e., that the transformation rules and strategies are able to pass any input term through the various canonical forms identified in the design.
In this section, we consider a fragment of the TOP design for the classloader. In particular we will restrict our attention to the first three canonical forms of the design described in Section 7.3.1. We will refer to this fragment of the design as our partial classloader design. The partial classloader design is implemented by the following transformation rules and strategies discussed earlier: resolve_info, resolve, relevant_cp, relevant, initial_offset, percolate_offset, calculate_offsets.

The sequential application of resolve, relevant, and calculate_offsets to a classfile will produce a classfile that is in Canonical Form 3. In the following sections we sketch how one might go about verifying the correctness of the partial classloader design as well as its implementation.

8.1 Design Verification: Eval and BCE

The approach to verification described below is based on viewing the Java Virtual Machine in terms of a semantic function that defines the meaning of each bytecode relative to a set of classfiles and a current state. We extend this idea to a framework in which a number of such semantic functions can be defined. In this framework, semantic functions are linked. The functionality common between them is explicitly identified and shareable. Our goal is to demonstrate an equivalence, modulo transformation, between the semantic function defining the JVM and the semantic function defining the SSP.

Let Eval be a class of semantic functions that take as arguments a set of bytecode class files \( \mathcal{C} \) and a program state \( s \) and return as output the next program state \( s' \). The program state records the values in program memory (the heap, frame, and opstack), registers, and the next bytecode instruction to be executed. Let \( \text{Eval}_{JV} \in \text{Eval} \) denote the semantic function that, given a set of classfiles \( \mathcal{C}_0 \) and a state, computes the next state in accordance with the JVM specification. Similarly, let \( \text{Eval}_{SSP} \in \text{Eval} \) denote the semantic function that, given a transformed set of classfiles \( \mathcal{C}_n = T(\mathcal{C}_0) \) (where \( T \) denotes the transformation implementation of the classloader) and a state, computes the next state in accordance with the SSP specification. We want to show that these two functions with their respective inputs compute the same values. This problem can be decomposed into a sequence of equivalences as follows. For each (intermediate) canonical form \( i \) produced by the classloader, we define a semantic function \( \text{Eval}_i \). To show that \( \text{Eval}_{JV}(\mathcal{C}_0, s) = \text{Eval}_{SSP}(\mathcal{C}_n, s) \), we will demonstrate that

\[
\text{Eval}_{JV}(\mathcal{C}_0, s) = \text{Eval}_1(\mathcal{C}_1, s) = \text{Eval}_2(\mathcal{C}_2, s) = \ldots = \text{Eval}_{SSP}(\mathcal{C}_n, s)
\]

Decomposing \( \text{Eval}_{JV}(\mathcal{C}_0, s) = \text{Eval}_{SSP}(\mathcal{C}_n, s) \) into a set of equivalences allows the proof to be constructed incrementally, reducing the complexity of the proof.

Let \( s' = \text{Eval}_{JV}(\mathcal{C}_0, s) \) for input state \( s \). In order for \( \text{Eval}_{JV} \) to correctly compute \( s' \), it must effect the actions required by the JVM specification for the particular bytecode instruction to be executed in \( s \). We define two sets of functions, \( \mathfrak{F}^0 \) and \( \mathfrak{F}^3 \) as follows. Let \( \mathfrak{F}^0 \) be the set of functions that define the semantics of Java bytecodes.
In other words, for each bytecode there exists one function in $\mathfrak{F}^b$ defining its semantics. Let $\mathfrak{F}^a$ be the set of auxiliary functions that may be used by the functions in $\mathfrak{F}^b$.

In the partial classloader example that we are considering, $\mathfrak{F}^a = \{ \text{info, access} \}$. The function info returns the resolved information for a constant pool index. For a standard classfile, this function traces through the indirection and returns the actual value required for computation. On the other hand, the function access, when given an index into the constant pool, simply returns the corresponding constant pool entry without tracing indirection.

In this discussion, we assume that the functions in $\mathfrak{F}^b$ may only access the constant pool through the functions in $\mathfrak{F}^a$. This means that the information associated with an entry necessary for properly executing bytecodes can only be obtained via the function info. Other information such as tags on constant pool entries and indexes are not needed other than to determine the information in an entry, and hence can be removed from $\mathfrak{C}_n$.

**Definition 1** An interpretation $I = \mathfrak{F}^a \cup \mathfrak{F}^b$ is the set of functions needed to compute a successor state given a set of classfiles and an initial state.

**Definition 2** Let $I_{JVM} = (\mathfrak{F}^a_{JVM} \cup \mathfrak{F}^b_{JVM})$ be the interpretation corresponding to the JVM.

**Definition 3** Let $I_{SSP} = (\mathfrak{F}^a_{SSP} \cup \mathfrak{F}^b_{SSP})$ be the interpretation corresponding to the SSP.

**Definition 4** Let $\mathcal{BCE}$ be a (byte code evaluator) function that takes an interpretation and produces a function $\text{Eval}_I \in \text{Eval}$. In particular, $\text{Eval}_{JVM} = \mathcal{BCE}(I_{JVM})$, and $\text{Eval}_{SSP} = \mathcal{BCE}(I_{SSP})$.

When defining the semantic functions $\text{Eval}_{JVM}$ and $\text{Eval}_{SSP}$, we require that $\mathfrak{F}^b_{SSP} = \mathfrak{F}^b_{JVM}$. Thus, all differences between these two semantic functions must be localized in $\mathfrak{F}^a$. Therefore what must be investigated is the relationship between the functions in $\mathfrak{F}^a_{JVM}$ and $\mathfrak{F}^a_{SSP}$ relative to the classloader.

Note that every function $\text{Eval}_I$ in the sequence from $\text{Eval}_{JVM}$ to $\text{Eval}_{SSP}$ is parameterized on classfiles of a distinct type. Let $\mathfrak{C}_{i-1}$ denote a set of classfiles having a type appropriate for $\text{Eval}_{i-1}$. The set of classfiles $\mathfrak{C}_i$ appropriate for $\text{Eval}_i$ is then derived from $\mathfrak{C}_{i-1}$ via a transformation strategy $T_i$ capable of transforming $\mathfrak{C}_{i-1}$ to $\mathfrak{C}_i$. If we define $M_i$ to be the mapping of functions in $\mathfrak{F}_{i-1}$ to the corresponding functions in $\mathfrak{F}_i$, then in order to show that $\text{Eval}_{i-1}(\mathfrak{C}_{i-1}, s) = \text{Eval}_i(\mathfrak{C}_i, s)$, it is necessary to show the following:

$$\left[\mathcal{BCE}(I_i)\right](s, \mathfrak{C}_i) = \left[\mathcal{BCE}(M_i(I_i))\right](T_i(s, \mathfrak{C}_i))$$

which is true if

57
∀(f_i ∈ \mathcal{F}_{i-1}) \exists (f_j ∈ \mathcal{F}_i) : f_j = M_i(f_i)

∀(f_i ∈ \mathcal{F}_i) : f_i(\mathcal{C}_i) = M_i(f_i)(T_i(\mathcal{C}_i))

From this we see that the various interpretations of BCE form the basis for formally defining the meaning of the canonical forms in our design as well as all intermediate forms of classfiles that are produced by the classloader. At one end of the spectrum, BCE produces a function capable of executing bytecodes with respect to “pure” Java classfiles. At the other end of the spectrum, BCE produces a function capable of executing bytecodes with respect to ROM image classfiles. For all interpretations, the functions produced by BCE compute the same results when applied to the appropriate classfiles. Thus, the goal of a transformation step is to produce classfiles that preserve bytecode equivalence.

Below we sketch various interpretations of \( \mathcal{F} \) = \{ info, access \}. We represent a constant pool as a list of constant pool entries, i.e., we abstract the constant pool to the extent that we can focus on the semantics of resolution and ignore the technical details of data representation. Furthermore, we abstractly define a constant pool entry as a list of data elements where the elements may be indexes or strings. In this setting, the information associated with a constant pool entry is a list of strings where each a string is ultimately the value of a utf8 entry. For example, in its unresolved form, a constant_pool_info entry is a list of two indexes (an index to a constant_pool_info entry, and an index to a constant_pool_info entry). In contrast, the information in a resolved constant_pool_info entry will be a list containing (1) a string denoting the name of a class, (2) a string denoting the name of the field, and (3) a string denoting the type of the field (i.e., the field descriptor).

Given this model of the constant pool and its entries, we now formally define the auxiliary functions info and access. We encapsulate each interpretation of the auxiliary functions \( \mathcal{F} \) in a structure where functions within the structure can be externally referenced using the traditional dot notation. Within a structure for a given interpretation, we define info and access in an equational manner using a syntax similar to the programming language ML. For the sake of readability, we pass a minimal number of parameters to each function.

Below we define three interpretations for info and access. While the definitions of these interpretations is not completely formal, we hope that they are rigorous enough to convince the reader that their complete formalization would not be particularly difficult.

\[ I_0: \]

\[ (\star \rightarrow \star) \]

\[^{10}\]The drawback of minimizing parameters is that we must assume that certain information such as the constant pool is globally available to auxiliary functions whenever needed. This is a small technical problem that can be easily fixed in a more complete treatment of the definitions.

58
info( Index(i)::es ) = info(access( i, constant_pool )) @ info(es)
info( String(s)::es ) = String(s)::info(es)
info( [] ) = []

access( 1, entry::entry_list ) = entry
access( index, entry::entry_list ) = access(index-1,entry_list)
access( index, [] ) = undefined

(* ----------------------------- *)
\[ \mathcal{I}_1 : \]

info( Index(i)::es ) = \mathcal{I}_0.info(Index(i)) @ info(es)
info( String(s)::es ) = String(s)::info(es)
info( [] ) = []

access( j, element::entry_list ) = if element = (j,entry) then entry
else access(j,entry_list)
access( j, [] ) = \mathcal{I}_0.access(j, constant_pool)

(* ----------------------------- *)
\[ \mathcal{I}_2 : \]

info( Index(i)::es ) = \mathcal{I}_1.info(Index(i)) @ info(es)
info( String(s)::es ) = String(s)::info(es)
info( [] ) = []

access( j, element::entry_list ) = if element = (j,k,entry) then entry
else access(j,entry_list)
access( j, [] ) = \mathcal{I}_1.access(j, constant_pool)

(* ----------------------------- *)

In interpretation \( \mathcal{I}_0 \), constant pool entries are abstractly referenced by their position in the list, and the information associated with a constant pool entry is obtained by resolving indexes within an entry until all that remains is a list of string values. The interpretation \( \mathcal{I}_1 \), is stacked on top of the definition of \( \mathcal{I}_0 \). \( \mathcal{I}_1 \) assumes that constant pool entries are tuples of the form \((\text{index}, \text{entry})\), where \text{index} is the abstract reference of the entry and the second element is a fully resolved entry as defined in \( \mathcal{I}_0 \). Note that if a particular element of the expected form cannot be found, the search is repeated using the functionality defined in \( \mathcal{I}_0 \). Similarly, if the information in an entry is not in its expected (i.e., resolved) form, a resolution call is made to the info function in \( \mathcal{I}_0 \).

In a similar fashion, the interpretation \( \mathcal{I}_2 \), is stacked on top of \( \mathcal{I}_1 \). The interpretation \( \mathcal{I}_2 \) differs from \( \mathcal{I}_1 \) only in that \( \mathcal{I}_2 \) expects constant pool entries to be triples of the form \((\text{offset}, \text{index}, \text{entry})\) where \text{offset} is the physical offset of the constant pool entry within
memory. In $I_2$ constant pool entries are accessed by their offset rather than their (abstract) index.

In the interpretations $I_0$, $I_1$, and $I_2$, if one strips away all references to previously defined interpretations, one is left with semantic functions that are suitable for constant pools that are in the appropriate canonical forms. The goal of transformation is to produce such canonical forms. Stacking of interpretations is needed to account for the incremental nature of transformation (intermediate forms are produced that are not canonical). Stacking of interpretations gives a well defined semantics to intermediate forms.

### 8.2 Basis for Verifying the Partial Classloader Design

Informally, the correctness of the partial classloader design can be argued as follows:

1. The representation of information in a direct fashion in constant pool entries is a refinement of the representation of the same information via indirection.
2. Explicit indexing enables entries to be removed from the constant pool while preserving abstract indexing.
3. All constant pool entries that are not referenced anywhere in the classfile can be removed from the constant pool.
4. Offsets can be computed for the remaining entries according to their specified sizes.

Let $T_1$, $T_2$, and $T_3$ respectively denote the following transformations:

1. Given a Java classfile $cf$, $T_1(cf)$ is a classfile in Canonical Form 1.
2. Given a classfile $cf'$ in Canonical Form 1, $T_2(cf')$ is a classfile in Canonical Form 2.
3. Given a classfile $cf''$ in Canonical Form 2, $T_3(cf'')$ is a classfile in Canonical Form 3.

From a formal standpoint, we want to show the following:

**Theorem 5** $\forall cf, \text{ state: } BcE[I_1](\text{state, } cf) = BcE[I_2](\text{state, } T_1(cf))$  

The proof of this is based on the assumption that information from an constant pool entry can only be obtained by calling the info function.

**Theorem 6** $\forall cf, \text{ state: } BcE[I_2](\text{state, } T_1(cf)) = BcE[I_2](\text{state, } T_2(T_1(cf)))$  

The proof of this is based on the fact that no bytecode has an index corresponding to entries that are not relevant.

60
Theorem 7 \( \forall cf, \text{state}: BCE[I_2](\text{state}, T_1(T_2((cf)))) = BCE[I_3](T_3(T_2(T_1(cf)))) \)

The proof of this is based on the fact that the sizes of constant pool entries in the ROM image are specified by the SSP.

8.3 Implementation Verification

In general, interpretations can be viewed as the pre- and post-conditions of transformation sequences that take a term from one canonical form to the next. Given an interpretation \( I_1 \) for a canonical form \( CF_1 \) and an interpretation \( I_2 \) for a canonical form \( CF_2 \), one must demonstrate that the transformation rules and strategies are able to transform any term in \( CF_1 \) to a corresponding term in \( CF_2 \). In this setting, the interpretations \( I_1 \) and \( I_2 \) determine how terms in \( CF_1 \) correspond to terms in \( CF_2 \).

For example, to show that the dynamic transformation rule \( \text{resolve}_\text{cp} \) together with the strategy \( \text{resolve} \) correctly transform an indexed classfile into canonical form 1 (as defined in our partial classloader design) we need to show that all index data are correctly replaced by string data throughout the classfile. Informally we argue that one can use the semantics of dynamic transformation rules to show that \( \text{resolve}_\text{cp} \) will produce transformation rules that, when applied, correctly replace constant pool index data found throughout the classfile with data that is presently associated with the constant pool entry having that index. Furthermore, it can be argued that an exhaustive application of these transformations (as is done by the \( \text{resolve} \) strategy) will remove all indirection from the constant pool.

9 Summary and Conclusion

In this chapter, we described high-consequence systems and argued that the ability to provide high assurance is one of the major risks faced in their development. Sufficient failures of high-consequence systems have been documented to support the allocation of resources in order to develop high-assurance systems. However, the willingness of management to devote sufficient resources is often clouded by short-term objectives such as return on investment. Another reason why management fails to allocate sufficient resources stems from a perception problem. It is difficult for humans to fully comprehend extremely large or extremely small numbers such as those used to define the reliability requirements for high-consequence systems. One might wonder if it matters (or how much it matters) that the system built has a reliability of 1-in-10^8 rather than 1-in-10^9.

Increasing the rigor in software development leads to the construction of more robust systems. The up-front costs associated with increased rigor often appear to be noticeably higher than the up-front costs associated with more traditional software development practices. However, increased expenditures early in a project are frequently rewarded by substantial savings later in the project (e.g., during the testing and maintenance phases). Furthermore, when developing a family of products, the cost of rigor can often be amortized over the product family, reducing the costs associated with rigor further.

61
In order to add rigor to software development, we proposed transformation-oriented programming (TOP) as a development paradigm. The approaches taken in TOP are based on equational reasoning. As such TOP lays the groundwork for the employment of formal verification to various development aspects, making TOP a suitable candidate for high-assurance software development. In TOP, the solution of a problem is captured by a TOP design and is realized by a TOP implementation. Transformation-oriented designs focus on passing an artifact (e.g., a formal specification) through a number of canonical forms. Transformation rules and strategies are realizations of transformation-oriented designs. We described in general terms a transformation system for implementing transformation-based designs, then we described the notation and capabilities of a specific transformation system called HATS. The remainder of the chapter was devoted to a TOP-based design and implementation of the static functionality of the Java Virtual Machine (JVM).

A  A Small Java Program

Jim McCoy developed the example below while he was working on the SSP project.

```java
// animal class example for inheritance
// jamecco 7/20/99
public class animal
{
private int Location;
protected int Feet;
protected int Position;
public static int START = 10;
public static int FORWARD = 11;
public static int BACKWARD = 12;
public static int STANDING = 11;
public static final int SITTING = 12;
public static final int LAYING = 13;
// constructors
animal()
{
Feet = 1;
Location = START;
Position = STANDING;
}
animal(int NumFeet)
{
Feet = NumFeet;
Location = START;
Position = STANDING;
```
} // the default way for all animals to stand
public void stand()
{
    Position = STANDING;
} // end of stand
// since Location is private even animal’s children can’t see it so we
// need a way for children to initialize it
public void setLoc(int NewLoc)
{
    if (NewLoc >= -25 & & NewLoc <= 25) // make sure it is safe
    {
        Location = NewLoc;
    }
    else
    {
        Location = -50; // otherwise put them in a known location
    }
} // end of setLoc
// the default way for animals to walk
// an animal object can modify Location directly but since Location is
// private none of animal’s children can see it or modify it directly
// this method provides a common interface for everything that is like
// an animal
public void walk(int Distance, int Direction)
{
    if (Position == STANDING & & Feet >= 2) // make sure the conditions are
    {
        // correct
        if (Direction == FORWARD) // and handle the different
            Location += Distance; // situations correctly
        else if (Direction == BACKWARD)
            Location -= Distance;
        else
            Location = START; // provide a default when things
    } // aren’t the way they should be
} // end of walk
// the default way for animals to count their feet
public int countFeet()
{
    return Feet;
} // end of countFeet
// the default way to find out what position an animal is in
public int getPos()
{
    return Position;
} // end of getPos
} // end of animal

B Java Classfile

Here we present the constant pool entries corresponding to the animal class in a human-readable form.

  animal
  0001
  java/lang/Object
  0003
  Location
  I
  Feet
  Position
  START
  FORWARD
  BACKWARD
  STANDING
  SITTING
  ConstantValue
  0000000C
  LAYING
  0000000D
  <init>
  ()V
  Code
  0012 0013
  0004 0015
  0007 0006
  0002 0017
  0009 0006
  0002 0019
  0005 0006
  0002 001B
  000C 0006
  0002 001D
  0008 0006
  0002 001F
C Resolved Classfile

This is the result of the three constant pool transformation steps we discussed. The numbers in the first column are hexadecimal offset values beginning with 4 as the base offset. The numbers in the second column are the hexadecimal abstract indexes that are originally used to reference constant pool entries.

0004 0002  = animal
0005 0004  = java/lang/Object
0006 000F  = 000000C
0007 0011  = 000000D
0008 0016  = java/lang/Object <init> ()V
000A 0018  = animal Feet I
000B 001A  = animal START I
000C 001C  = animal Location I
000D 001E  = animal STANDING I
000E 0020  = animal Position I
000F 002D  = animal FORWARD I
References


